《Java编程思想》的作者Bruce Eckel认为，Kotlin或将取代Java

阿里巴巴资深程序员呕心沥血之作，揭秘Kotlin编程的精华

全面涵盖Kotlin基础语法、进阶实战技巧和项目案例开发等实用内容

Kotlin 从入门到进阶实战

陈光剑 编著



清华大学出版社

Kotlin 从入门到进阶实战

陈光剑　编著



清华大学出版社
北　京

内 容 简 介

本书从 Kotlin 语言的基础语法讲起，逐步深入到 Kotlin 进阶实战，并在最后配合项目实战案例，重点介绍了使用 Kotlin+Spring Boot 进行服务端开发和使用 Kotlin 进行 Android 应用程序开发的内容，让读者不但可以系统地学习 Kotlin 编程的相关知识，而且还能对 Kotlin 应用开发有更为深入的理解。

本书分为 14 章，涵盖的主要内容有 Kotlin 简介，Kotlin 语法基础，类型系统与可空类型，类与面向对象编程，函数与函数式编程，扩展函数与属性，集合类，泛型，文件 I/O 操作、正则表达式与多线程，使用 Kotlin 创建 DSL，运算符重载与约定，元编程、注解与反射，Kotlin 集成 Spring Boot 服务端开发，使用 Kotlin 进行 Android 开发。

本书内容通俗易懂，案例丰富，实用性强，特别适合 Kotlin 语言的入门读者和进阶读者阅读，也适合 Android 程序员、Java 程序员等其他编程爱好者阅读，还适合作为相关培训机构的教材。

图书在版编目（CIP）数据

Kotlin 从入门到进阶实战 / 陈光剑编著. —北京：清华大学出版社，2018
ISBN 978-7-302-50872-4

Ⅰ. ①K…　Ⅱ. ①陈…　Ⅲ. ①JAVA 语言－程序设计　Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2018）第 181509 号

责任编辑： 杨如林
封面设计： 欧振旭
责任校对： 徐俊伟
责任印制： 李红英

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn, http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者： 三河市金元印装有限公司
经　　销： 全国新华书店
开　　本： 185mm×260mm　　**印　　张：** 17　　**字　　数：** 428 千字
版　　次： 2018 年 9 月第 1 版　　**印　　次：** 2018 年 9 月第 1 次印刷
定　　价： 69.80 元

产品编号：080566-01

前　　言

当下，互联网、大数据和云计算迅猛发展，数以百万计的应用程序在服务器和移动端运行。这些应用程序的开发语言有很大一部分是用软件界已经流行了 20 年之久的主力编程语言 Java 编写的。

毫无疑问，Java 语言历史悠久，影响力巨大。历经 20 多年的发展，它已经成为一门非常成熟的编程语言，性能强大而稳定。Java 虚拟机 JVM 的生态也繁荣昌盛，经久不衰。但 Java 也背负着历史的包袱，如它有空指针、语法啰嗦和不支持一等函数等缺点。如果用一辆汽车来比喻编程语言，Java 拥有一个高效而可靠的发动机，但其防抱死刹车系统和动力转向系统却不是那么可控。Java 语言在使用时需要小心检查可能出现的空指针，还要处理异常、重复生成冗长而单调的样板代码行等问题。

对于开发人员而言，编程语言的防危性（safety）和安全性（security）是至关重要的。要是有一门语言既能继承 Java 的所有优点及其强大而完备的生态库，又能更加简单、安全和可控，那真是再好不过了。我们很高兴地看到，Kotlin 就是一门这样的语言。

目前，图书市场上 Kotlin 相关图书还很少，尤其是实用性强的书更是凤毛麟角。为了帮助广大的编程人员系统地学习这门开发语言，笔者编写了本书。本书从 Kotlin 语言的基础语法讲起，逐步介绍了 Kotlin 的扩展函数、一等函数支持、Lambda 表达式、强大的 DSL 支持、运算符重载与约定、无编程、注解与反射等特性，并配合项目实战案例，详细介绍了使用 Kotlin+Spring Boot 进行服务端开发和使用 Kotlin 进行 Android 应用程序开发的内容。通过阅读本书，读者不但可以系统地学习 Kotlin 编程的相关知识，而且还能对 Kotlin 应用开发有更为深入的理解。

本书特色

1．内容全面，讲解由浅入深，符合学习规律

本书内容涵盖了 Kotlin 语言的基础语法和大部分最常用的核心知识点和开发技巧，还详细介绍了两个实用性很强的项目开发案例。讲解遵循由浅入深、循序渐进的原则，让读者的学习曲线更加平滑。这样的内容梯度安排和讲解，符合读者的编程语言学习规律，可以取得较好的学习效果。

2．图文并茂，讲解生动有趣，阅读起来不枯燥

技术学习，有时一图胜千言。本书在介绍知识点时尽量给出简单易懂的图示帮助读者理解，这使得整个学习过程变得简单、有趣。

3．用代码示例引导学习，可以大大提高动手编程能力

本书非常注重内容的实用性和可操作性，书中重点介绍的知识点都给出了大量代码示例，并且对代码做了详细的注释和讲解，这样可以大大提高读者实际动手编程的能力。

4．偏重于实战讲解，不涉及不常用的知识

相比笔者的另外一本书《Kotlin 极简教程》，本书内容更加偏重于 Kotlin 编程实战讲解。书中对于 Kotlin 基础知识和语言特性的讲解更加精简，重点突出；而对于编程实战中不常用的一些内容不做过多介绍，比如没有介绍目前不常用的 Kotlin Native 和实验阶段的协程（Coroutine）两个专题；但增加了在编程实践中较为常用的元编程、注解与反射，运算符重载与约定两章的内容。

5．项目案例实用性强，可以提高项目开发水平

本书最后两章配合项目实战案例，详细介绍了使用 Kotlin+Spring Boot 进行服务端开发和使用 Kotlin 开发 Android 应用程序的相关内容。这两个项目案例可以带领读者体验实际的 Kotlin 应用开发，可以大幅度提高读者的项目实战开发水平。

本书内容

第 1 章主要介绍了 Kotlin 编程语言的基本特性、编程哲学、学习工具，以及为什么要学 Kotlin 和 JVM 语言生态等内容。

第 2 章主要介绍了 Kotlin 语法基础，主要内容包括变量和标识符、关键字与修饰符、流程控制语句、操作符与重载、包声明等内容。

第 3 章主要介绍了 Kotlin 的类型系统、可空类型、安全操作符、特殊类型、类型检测与类型转换等内容。

第 4 章主要介绍了 Kotlin 的类与面向对象编程，包括声明类、抽象类与接口、object 对象、数据类、注解、枚举和内部类等内容。

第 5 章主要介绍了 Kotlin 函数式编程，包括声明函数、Lambda 表达式、高阶函数及 Kotlin 中的特殊函数等内容。

第 6 章主要介绍了 Kotlin 扩展函数与属性，以及扩展函数的实现原理和扩展中的 this 关键字。

第 7 章主要介绍了 Kotlin 集合类，包括常用的 3 种集合类、不可变集合类、创建集合类、遍历集合中的元素、映射函数、过滤函数、排序函数和元素去重等内容。

第 8 章主要介绍了 Kotlin 的泛型，包括为何引入泛型、泛型接口、泛型类、泛型函数、类型上界、协变与逆变、out T 与 in T、类型擦除等内容。

第 9 章主要介绍了 Kotlin 语言的文件 I/O 操作、网络 I/O 操作、执行 Shell 命令、正则表达式和多线程编程等相关内容。

第 10 章主要介绍了怎样使用 Kotlin 语言创建 DSL，包括什么是 DSL、Kotlin 的 DSL 特性支持，同时实现了一个集合类的流式 Kotlin DSL 实例及一个 SQL 风格的集合类 DSL 实例。

第 11 章主要介绍了 Kotlin 的运算符重载与约定，包括什么是运算符重载、重载二元算术运算符、重载自增自减一元运算符、重载比较运算符及重载计算赋值运算符等内容。

第 12 章主要介绍了 Kotlin 元编程、注解与反射的相关内容，包括元编程简介、声明注解、使用注解、处理注解、反射、类引用、函数引用、属性引用、绑定函数、使用反射获取泛型信息等内容。

第 13 章介绍 Kotlin 集成 Spring Boot 服务端开发，首先用 Spring Boot 快速开发一个 Restful Hello World 示例，然后给出了一个完整的图片爬虫 Web 应用项目案例。

第 14 章介绍如何使用 Kotlin 进行 Android 开发，首先给出了一个简单的 Kotlin 版本的 Hello World Android 示例程序，然后详细介绍了用 Kotlin 开发一个电影指南 Android 应用程序综合项目案例。

本书读者对象

- Kotlin 入门人员；
- Kotlin 进阶开发人员；
- Android 程序员；
- Java 程序员；
- 其他编程爱好者；
- 相关培训机构的学员。

本书源程序获取方式

本书涉及的源代码需要读者自行下载。请登录清华大学出版社网站 www.tup.com.cn，搜索到本书页面，在页面上找到“资源下载”栏目，然后单击“课件下载”或者“网络资源”按钮即可下载。

作者与致谢

笔者现就职于阿里巴巴集团，曾经参与了多种平台工具的开发，主要使用 Java、Android、Scala、Groovy 和 Kotlin 等语言或工具进行领域建模、架构设计和工具开发等，积累了大量经验。

感谢在本书写作过程中提供过帮助的各位朋友！也感谢在本书出版过程中提供过帮助的各位编辑，没有你们的付出，本书就不会顺利和读者见面！最后感谢各位读者选择了本书，祝你们学习愉快！

虽然笔者对书中所述内容都尽量核实，并多次进行文字校对，但因时间所限，加之水平所限，书中可能还存在疏漏和错误，敬请广大读者批评指正。联系 E-mail：bookservice2008@163.com。

陈光剑
于杭州

目　　录

第 1 章　Kotlin 是什么

Kotlin 是一种非研究性并且非常务实的工业级编程语言，它的使命就是帮助程序员解决实际工程实践中的问题。使用 Kotlin 语言让 Java 程序员的工作变得更轻松，Java 语言中的那些空指针错误、浪费时间的冗长的样板代码、啰嗦的语法限制等，在 Kotlin 语言中统统消失。Kotlin 语言简单、务实，语法简洁而强大，安全且表达力强，极富生产力。

本章首先简单介绍 Kotlin 语言的发展历史和语言特性，然后简述为什么要学习 Kotlin 语言，最后简要介绍 JVM 语言家族。

1.1　初识 Kotlin

Kotlin 是一种基于 JVM 的静态类型编程语言。Kotlin 从开始推出至今已经有 7 年，2016 年官方正式发布了首个稳定版本。Kotlin 发展简史如下：

- 2011 年 7 月，JetBrains 推出 Kotlin 项目。
- 2012 年 2 月，JetBrains 以 Apache 2 许可证开源此项目。
- 2016 年 2 月 15 日，Kotlin v1.0（第 1 个官方稳定版本）发布。
- 2017 Google I/O 大会上，Kotlin“转正”。

Kotlin 具备类型推断、多范式支持、可空性表达、扩展函数、模式匹配等诸多下一代编程语言特性。

Kotlin 的编译器 kompiler 可以被独立出来并嵌入到 Maven、Ant 或 Gradle 工具链中。这使得在 IDE 中开发的代码能够利用已有的机制来构建，可以在新环境中自由使用。

让我们从 Hello World 开始。与 C、C++、Java 语言一样，Kotlin 程序的入口点是一个名为 main()的函数，它传递一个包含任何命令行参数的数组。代码示例如下：

```
package com.easy.kotlin                 //(1)
fun main(args: Array<String>) {         //(2)
    val name = "World"
    println("Hello,$name!")             //(3)
}
```

上面的代码简单说明如下。

（1）：Kotlin 中包 package 的使用与 Java 基本相同。有一点不同的是 Kotlin 的 package 命名可以与包路径不同。

（2）：Kotlin 变量声明 args:Array 类似于 Pascal，先写变量名 args，冒号隔开，再在后面写变量的类型 Array。与 Scala 和 Groovy 一样，代码行末尾的分号是可选的，在大多

数情况下，编译器根据换行符就能够推断语句已经结束。Kotlin 中使用 fun 关键字声明函数（方法），充满乐趣的 fun。

（3）：Kotlin 中的打印函数是 println()（虽然背后封装的仍然是 Java 的 System.out.println() 方法）。Kotlin 中支持字符串模板$name，如果是表达式，则使用${expression}语法。

1.2　语 言 特 性

人们为什么喜欢 Kotlin？Kotlin 为什么值得我们去学习？下面是一个不完全的清单列表。

- 与 Java 及 JVM 的完全互操作性；
- 多平台：适合 Android、浏览器（JavaScript）和本地系统编程（native）；
- 语法简洁不啰嗦（便于学习）；
- 富于表现力和高效的生产力；
- 支持类型推断。例如，我们可以只写 val number=23，编译器会推断这是一个 Int；
- 可以使用数据类（data class）以极简的方式创建 POJO；
- 运算符重载相当简单；
- 快速、方便地扩展内置类、自定义类的函数与属性；
- 区分可空类型和不可空类型。直接在编译期语法层面检查可空类型，提供空安全保障；
- Kotlin 含有功能丰富的集合类 Stream API；
- 集成扩展了简单实用的文件 I/O、正则匹配、线程等工具类；
- 提供了实用强大的函数式编程支持：一等函数支持，Lambda 表达式、高阶函数等；
- 能够轻松、方便地创建 DSL；
- 使用更加轻量级的协程进行并发编程；
- IntelliJIDEA 开发工具的一等支持；
- Android 开发有 Android Studio 3 内置原生支持；
- 提供的 Anko 库（https://github.com/Kotlin/anko）使得 Android 开发速度更快，充满更多的乐趣等。

Kotlin 的优势是既有 Java 的完整生态（Kotlin 完全无缝使用各类 Java API 框架库），又有现代语言的高级特性（语法糖）。

Kotlin 语言的设计初衷之一是为了 JetBrains 团队内部使用，旨在帮助公司降低成本。用过 IntelliJ IDEA 的程序员都知道 JetBrains 团队的出品皆是良品。毫无疑问，Kotlin 的设计是务实的。发展和促进 Kotlin 的好处大于其成本，在这个过程中，Kotlin 已经演变成了一个 JetBrains 的效率工具，其显著的务实特性吸引了一大批 Java 程序员，并成为了 JetBrains 工具生态系统中重要的一员。

在未来几年内，Kotlin 有望成为主要的非 Java 的 JVM 语言，甚至有一天成为下一个“Java”语言。可以预测的是，Kotlin 将大大提升整个 Java 互联网开发者的效率和质量。

Kotlin 语言的特性可以简单概括为以下几方面。

1. 实用主义（Pragmatic）

务实、注重工程实践性。我们经常会听到人们说编程是数学，或者是工程，是艺术，是科学，这些说法都是很有道理的。Kotlin 是一门偏重工程实践、编程上有极简风格的语言。

2. 极简主义（Minimalist）

Kotlin 语法简洁优雅不啰嗦，类型系统中一切皆是引用（reference）。

3. 空安全（Null Safety）

Kotlin 中有一个简单完备的类型系统来支持空安全。

4. 多范式（multi-paradigm）

Kotlin 同时支持 OOP 与 FP 编程范式。各种编程风格的组合可以让我们更加直接地表达算法思想和解决问题的方案，可以赋予我们在思考上有更大的自由度和灵活性。

5. 可扩展

Kotlin 可直接扩展类的函数与属性（extension functions & properties）。这与我们在 Java 中经常写的 util 类是完全不一样的体验！Kotlin 是一种非常注重用户体验的语言。

6. 高阶函数与闭包（higher-order functions & closures）

Kotlin 的类型中，函数类型（function type）也是一等类型（first class type）。在 Kotlin 中可以把函数当成值进行传递，这直接赋予了 Kotlin 函数式编程的特性，使用 Kotlin 可以写出一些非常“优雅”的代码。

7. 支持快速实现DSL

有了扩展函数、闭包等特性的支持，使用 Kotlin 实现一个 DSL 将会相当简单、方便。

1.2.1　Kotlin 与 Java 完全互操作

Kotlin 是基于 JVM 平台的静态编程语言，同时在设计之初就把与 Java 的互操作性当作重要目标。正如官方网站所宣传的那样：100% interoperable with Java and Android。下面我们举个简单例子来展示 Kotlin 中使用 Java 的 ArrayList 类与使用 JUnit 测试框架进行单元测试。代码示例如下：

```
fun getArrayList(): List<String> {     //(1)函数声明
    val arrayList = ArrayList<String>() //(2)Kotlin 中直接调用 Java 的 API
    arrayList.add("A")
    arrayList.add("B")
```

```
    arrayList.add("C")
    return arrayList
}
```

代码说明如下。

（1）：声明了一个返回 List<String>的函数，我们看到在 Kotlin 中使用 fun 关键字来声明函数。

（2）：创建了一个 ArrayList<String>对象，我们可以看到，在 Kotlin 中创建对象不再使用 new 关键字了，尖括号里面的 String 是泛型信息。该语法与 Java 语言基本类似。关于集合类与泛型的相关内容，将在第 7 章和第 8 章中具体介绍。

下面使用 JUnit 框架进行单元测试。代码如下：

```
package com.easy.kotlin                         //(3)包声明

import org.junit.Assert
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4                 //(4)导入 JUnit 的类

@RunWith(JUnit4::class)     //(5) 直接使用 Java 生态库 JUnit 中的注解@RunWith
class FullJavaInteroperabilityTest {
    @Test                                       //(6)标记这是一个测试方法
    fun test() {
        val list = getArrayList()               //(7)调用被测试函数
        Assert.assertTrue(list.size == 3)       //(8)断言
    }
}
```

代码说明如下。

（3）：是包声明，使用 package 关键字。

（4）：使用 import 导入 JUnit4 类。

（5）：Kotlin 中使用@RunWith 注解，方式与 Java 语法类似。注解中的参数是 JUnit4::class，是 JUnit4 类的引用。我们将在第 12 章中介绍注解与反射。

（6）：使用 JUnit 的@Test 注解来标注这是一个测试方法。

（7）：调用被测试函数 getArrayList()。

（8）：使用 JUnit 的 Assert 类的 API 进行断言操作。

1.2.2 扩展函数与扩展属性

扩展函数与扩展属性的“好玩”之处在于，可以在不修改原来类的条件下自定义函数和属性，使它们表现得就像是属于这个类一样。例如，我们给 String 类型扩展一个返回字符串首字母的 firstChar()函数，代码如下：

```
fun String.firstChar(): String {    //给 String 类扩展一个 firstChar()函数
    if (this.length == 0) {         //这里的 this 代表调用者对象
        return ""
    }
    return this[0].toString()       //返回下标为 0 的字符并转成 String 类型
}
```

然后就可以在代码中直接这样调用该函数：

```
"abc".firstChar()                //调用我们自定义的扩展函数
```

代码显得相当简洁。

1.2.3　不可空类型与空安全

使用 Kotlin 编程比 Java 更加安全，至少在空指针问题上写起代码来会更加“开心”。Kotlin 中引入了不可空类型与可空类型来明确声明一个变量是否可能为 null，同时在编译期通过类型是否匹配来检查空指针异常，大大降低了空指针异常出现的概率。同时，Kotlin 还提供了 Elvis 操作符、安全调用符等极简的语法格式，使开发者从 Java 的 null 防御式编程中被解放出来。例如下面的这段代码：

```
>>> var a = "abc"   //声明一个字符串,编译器会默认推断变量 a 的类型为不可空的 String
>>> a = null        //不可空类型不能赋值为 null
error: null can not be a value of a non-null type String
a = null
    ^
```

当声明了不可空类型 String 的变量 a 后，在后面使用变量 a 的代码中就不能给 a 赋值为 null。如果给 a 赋值 null，编译器会直接报错。而这个时候，如果我们想声明一个可空的 String 类型，可以这样写：

```
var b:String? = "abc"   //声明一个可空的 String?类型
```

但是这个时候，如果想调用变量 b 的方法，就不能直接像下面这样写了：

```
>>> b.length        //可空类型不能直接调用方法，需要使用安全调用符?.或者断言调用!!.
error: only safe (?.) or non-null asserted (!!.) calls are allowed on a
nullable receiver of type String?
b.length
 ^
```

我们可以看到，上面的代码运行报错：

```
可空类型 String? 只有使用安全调用符(?.)和非空断言调用符(!!)才允许调用其方法。
```

如果在 IDE 中会直接提示报错，编译不通过。上面的代码可以使用安全调用符进行如下改写：

```
>>> b?.length   //使用安全调用符
3
>>> b=null
>>> b?.length   //null 对象使用安全调用符访问 length 属性，直接返回 null
null
```

这个问号确实非常简洁易懂，同时能够时刻提醒我们：这个调用者有可能是 null 的。这个语法明显比 Java 8 中引入的 Optional<String>更加简单、直接。

1.2.4 一等函数支持

在 Kotlin 中函数是第一等类型（fist class）：我们可以将函数像值一样传递，函数可以作为另一个函数的返回值。我们通常称之为“一等函数（first order functions）”支持。例如，下面是一个把函数作为参数传递给函数的 Lambda 表达式的例子：

```
>>> val list = listOf(1, 2, 3, 4, 5, 6, 7) //声明一个不可变的 List
>>> list.filter{ it%3!=0 }
                    //调用 filter 函数，传入一个 Lambda 表达式{ it%3!=0 }作为参数
[1, 2, 4, 5, 7]
```

其中，{it%3!=0}是一个 Lambda 表达式，它判断元素是否能够被 3 整除。如果满足此条件就留下该元素，否则过滤掉。关于函数式编程的内容将在第 5 章中介绍。

1.2.5 智能类型推断

在上面的诸多例子中，可以看到在声明变量的时候并没有显式指定它的类型。Kotlin 编译器会自动推断出其类型。

上面介绍的只是 Kotlin 诸多优秀特性中的一部分，更多内容且看后面的章节讲解。

1.3 编 程 哲 学

“我们认为 Kotlin 的定位是一种现代化工业语言：它专注于代码重用和可读性的弹性抽象，以及面向早期错误侦测和明确捕获维护与清理的意图这些问题的静态类型安全性。Kotlin 最重要的使用场景之一是对于一个庞大的 Java 代码库，其开发者需要一个更棒的语言：你能够将 Java 和 Kotlin 自由混合，迁移可以是渐进式的，不需要一下子对整个代码库进行改变。”

“Kotlin 旨在成为一种面向工业的面向对象语言，而且是一种比 Java 更好的语言，但仍然可以与 Java 代码完全互操作，允许企业逐步从 Java 迁移到 Kotlin。”

——Andrey Breslav，Kotlin 创始人

编程的真正问题在于，如何把人类脑子里对问题的解决方案“具化”到机器世界，而这个“具化”的过程正是编程语言所要表达的东西。如何富有表现力并且安全简洁地表达，这是所有编程语言所要解决的问题。让人类能够尽可能“自然地”和计算机进行沟通交流，这一直是促使人们提高编程语言抽象层次的主要目标之一。很显然的一个事实就是，与用机器语言写的低层次结构代码相比，用编译语言写成的高层次结构代码更接近于人类进行思考时所用的概念。

Kotlin 设计了一个“归一化”的类型系统（一切类型皆是引用类型），纯天然地设置了一道空指针的屏障，使得 Kotlin 比 Java 更加安全可靠。Kotlin 还引入了类型推断、一等

支持函数式编程、Lambda、高阶函数、类的扩展函数与属性、DSL 等诸多特性，使得我们可以编写简单且高效的代码，更加专注地投入到业务逻辑的实现上。

优秀的程序员当然会选择使用 Kotlin 这些更加先进的特性，因为它们有助于更直接地表达观点，而且也没有额外的开销，何乐而不为呢？

1.4 学 习 工 具

工欲善其事，必先利其器。本节我们简单介绍一下学习 Kotlin 的工具平台。

1.4.1 云端 IDE

如果你想快速体验一下 Kotlin，只需要通过浏览器打开云端 IDE，网址为 https://try.kotlinlang.org/，如图 1-1 所示。

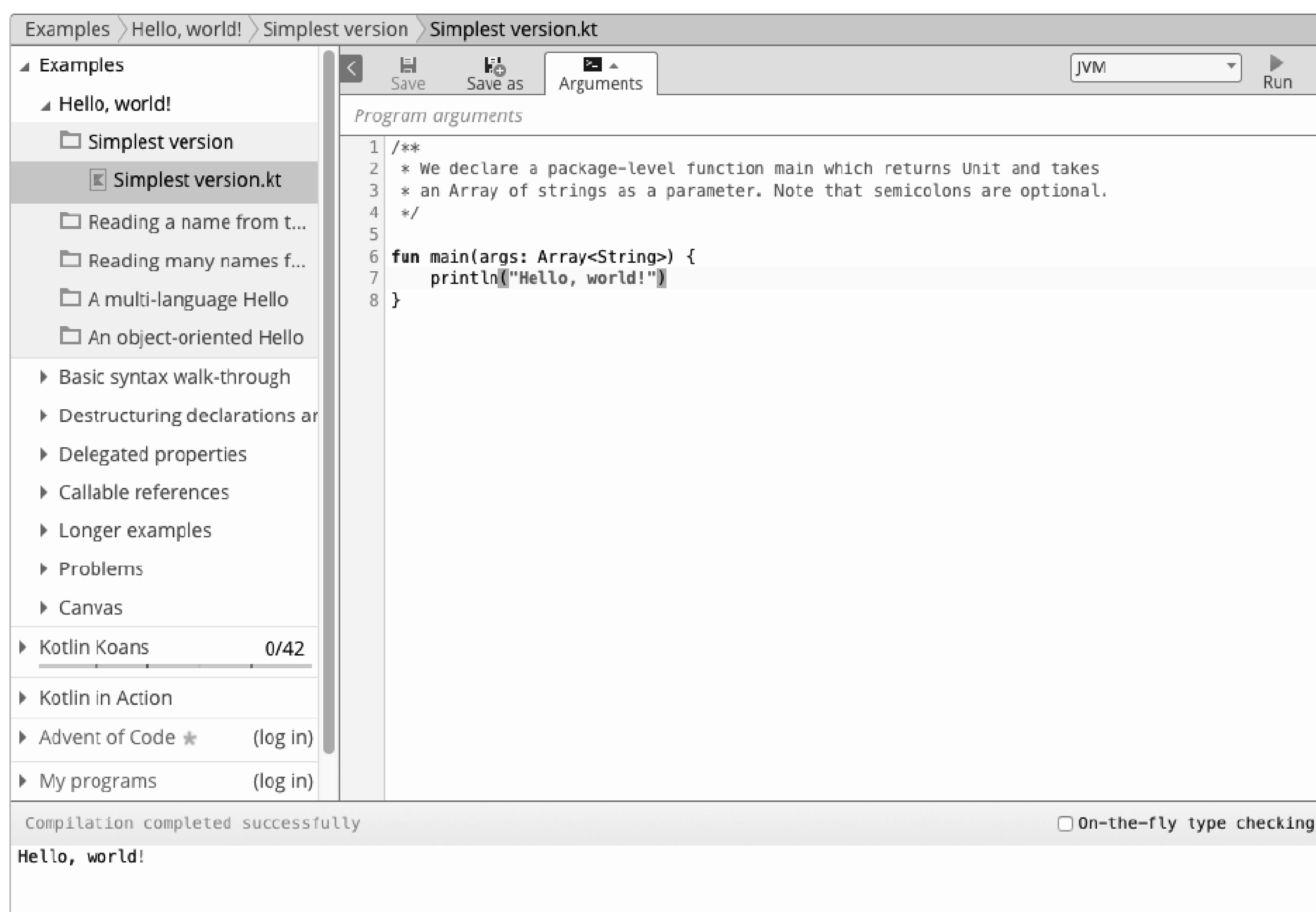


图 1-1　Kotlin 云端 IDE

在这里可以快速感受 Kotlin 语言到底是什么样的，但是这里不支持代码智能提示及自动补全等功能。

1.4.2 命令行 REPL

有时候我们并不需要打开 IDE 来做一些事情。打开 IDE 是件很麻烦的事情，在某些场景下，开发者比较喜欢命令行。

使用命令行环境，我们可以方便地使用 Kotlin REPL（Read-Eval-Print-Loop，交互式编程环境）。REPL 可以实时编写 Kotlin 代码，并查看运行结果。通常，REPL 交互方式可以用于调试、测试及测试某种效果。

如果你想在本地快速测试一个简短的 Kotlin 代码，可以使用命令行 REPL。Kotlin 是运行在 JVM 环境下的语言。首先我们要有 JDK 环境（此处省略 Java 环境配置）。

目前，Kotlin 正式发布的最新版本是 1.1.50。首先去下载 Kotlin 运行环境安装包：https://github.com/JetBrains/kotlin/releases/download/v1.1.50/kotlin-compiler-1.1.50.zip。

解压完 kotlin-compiler-1.1.50.zip，放到相应的目录下。然后配置系统环境变量：

```
export KOTLIN HOME=/Users/jack/soft/kotlinc
export PATH=$PATH:$KOTLIN_HOME/bin
```

执行 source ~/.bashrc，在命令行输入 kotlinc，即可进入 KotlinREPL 界面。

```
$ kotlinc
Welcome to Kotlin version 1.1.50 (JRE 1.8.0 40-b27)
Type :help for help, :quit for quit
>>> println("Hello,World!")     //在 REPL 中直接打印"Hello,World!"
Hello,World!
>>> import java.util.Date       //在 REPL 中导入需要使用的 Java 中的 Date 类
>>> Date()                      //创建一个 Date 对象，输出当前的时间
Wed Jun 07 14:19:33 CST 2017
```

1.4.3 使用 IDEA

如果想有学习 Kotlin 的相对较好的体验，那么建议读者不要使用 eclipse。毕竟 Kotlin 是 JetBrains 家族的“亲儿子”，跟 IntelliJ IDEA 是“血浓于水”啊。

我们使用 IDEA 新建 Kotlin Gradle 项目，选择 Java，Kotlin(Java)框架支持，如图 1-2 所示。

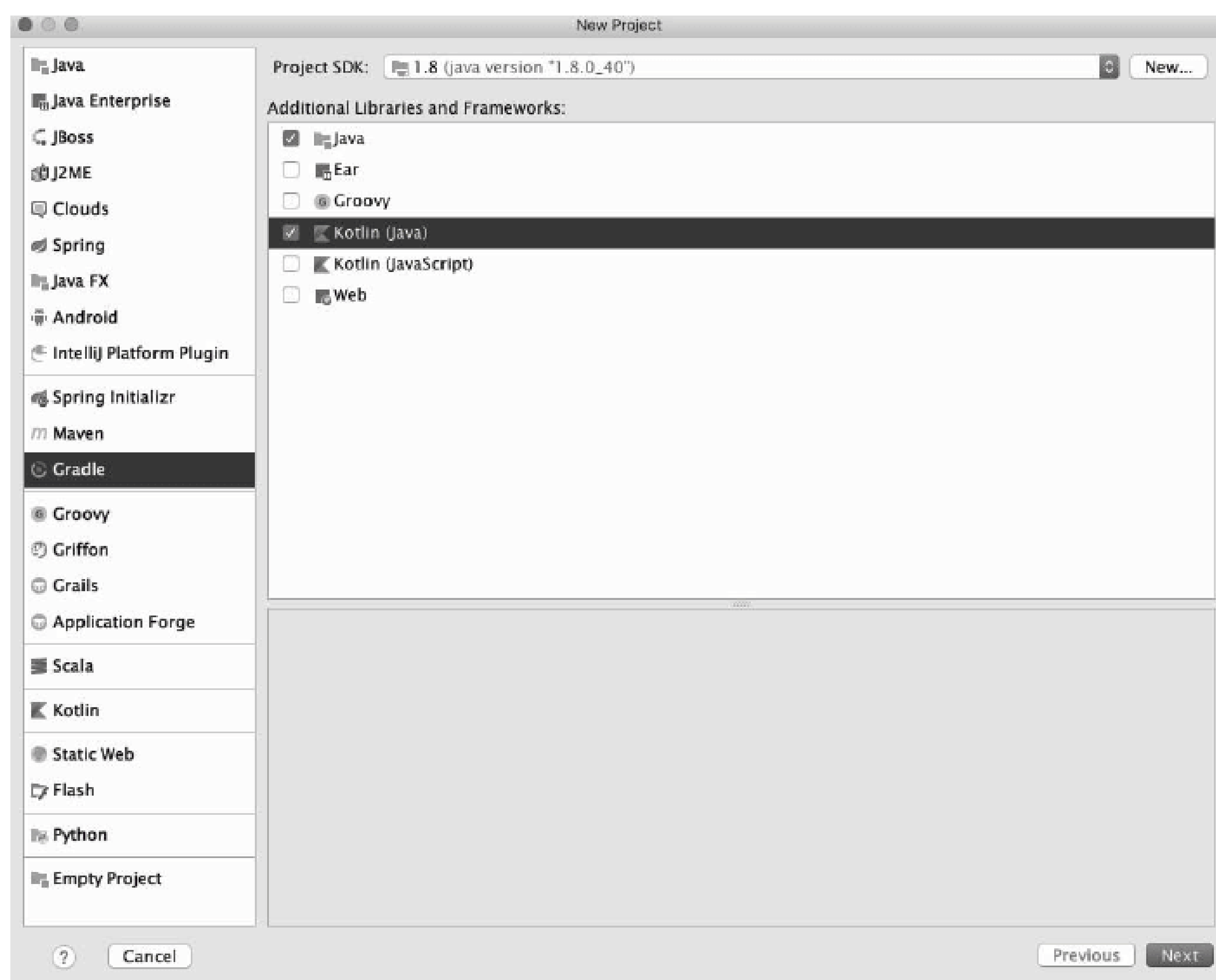


图 1-2 使用 IDEA 新建 Kotlin Gradle 项目

新建完项目后，我们写一个 HelloWorld.kt 类，代码如下：

```
package com.easy.kotlin

import java.util.Date                  //(1) 导入 Date 类
import java.text.SimpleDateFormat      //(2) 导入 SimpleDateFormat 类

fun main(args: Array<String>) {
    println("Hello, world!")           //打印函数
    println(SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(Date()))
                                   //(3) 直接使用 Java 中的 API，语法跟 Java 类似
}
```

代码说明如下。

（1）：导入 Java 中的 Date 类。

（2）：导入 Java 中的 SimpleDateFormat 类。

（3）：直接使用 Java 中的 API。Kotlin 调用 Java 的语法对我们来说很熟悉了，直接运行 HelloWorld.kt，输出结果如下：

```
Hello, world!
2017-05-29 01:15:30
```

1.5　为什么要学 Kotlin

现在的编程语言已经足够多了，为什么我们还需要更多的语言？Java 已经足够强大了，为什么我们还需要 Kotlin、Scala 这样的语言呢？

其实，如果我们仔细想想，会发现这个问题本身的逻辑就不成立。例如，我们能这样说吗——煎鸡排已经足够好吃了，为什么我们还要去吃煎牛排呢？

从最早的机器语言（01 机器码，汇编语言）到高级语言（LISP、BASIC、Pascal、C、C++、Java、Haskell 等），再到现代编程语言（Go、Swift、Scala、Kotlin 等），编程语言的演化过程可谓是百花齐放、百家争鸣。

最早的编程语言是 01 机器码（Machine Code），那个时候的程序员要会用 0 和 1 表示一切！

后来人们可以把一些常用的指令操作单独抽象出来，用特定的关键字来映射 01 机器码序列，而这就是汇编语言，这可以算作是编程语言过程中的第一次抽象封装。也许汇编语言的主要意义不在于它与机器语言之间并不显著的差别，而是这样的一种思路：程序完全可以在不同的层次上编制！人们可以用机器语言写一个“翻译程序”，从而可以在一个更高层次上进行编程。

后来汇编语言用久了，人们也逐渐发现了使用汇编语言的缺点：可移植性差。汇编代码中是大量的字节指令码，而且必须一步步地告诉计算机每一步要怎么做，如果其中的一个步骤出错，那么执行结果将是程序员们意想不到的！使用汇编语言编程，极易在子程序调用过程中导致寄存器内容错误，而且调试程序也很困难。程序在正常运行时，我们基本不会过多地关心和想象它的活动结构和层次空间。只有当程序出现 bug 或者崩溃的时候，

我们才会在不同层次上思考和想象程序运行的具体细节。而这其中的出错信息将变得至关重要。例如，一个除法操作，遇到除数为 0 的情况程序将暂停运行，并把错误抛给程序员。下面是不同层次上的 debug 信息：

机器语言层：程序运行异常终止于 11110000010001001 地址；

汇编语言层：程序运行异常终止于 DIV 指令；

编译语言层：程序运行异常终止于代码行 256: (a+b)/c 处。

上面的信息中，显而易见的是层次越高，越容易被人类大脑所理解。

在高级语言中，所有参数都必须严格匹配其类型，这样就不会出现寄存器内容错误的情况。高级语言就是为了解决汇编语言的这些问题进行的更高一层的抽象与封装。这层封装就是编译器。编译器所要解决的问题就是如何构造一个系统，使它可以接收当前层次的描述，然后从中生成另一个层次上的描述。通常来说，设计一门语言相对容易，而实现这门语言的编译器则是比较复杂的。编译器制定了一系列的协议规范、语法规则等，只要程序员按照这个协议规范来编程，编译器就可以将高级语言的源代码翻译成对应 CPU 指令集上的汇编语言代码。高级语言不要求程序员掌握计算机的硬件运行原理，只要写好上层代码即可。著名的高级语言有 BASIC、Fortran（公式翻译）、COBOL（通用商业语言）、C、Pascal（结构化编程语言）、ADA（通用程序设计语言）等。

尽管 C 语言（1972，Dennis MacAlistair Ritchie，启发语言有 B 语言、汇编、ALGOL68 等）已经足够普及且非常强大，但是之后还是出现了针对 C 语言进行改进和功能扩展的新语言——C++语言（1979，Bjarne Stroustrup）。C++语言集成了 C 语言的特性，然后加入了面向对象程序设计的特性支持。和汇编语言不同的是，C 语言的语句和机器语言的指令之间不再是简单的一一对应关系，不过毫无疑问的是，仍然存在从 C 语言代码到机器语言代码的映射关系，但是这种关系要比从汇编语言到机器语言之间的关系复杂多了。而完成这个映射过程翻译的程序，我们就称之为“编译器”。

而 C/C++语言最大的一个问题就是“一切都会尖叫着停止”，因为它们使用了直接操纵内存的指针。一旦因为使用指针而出现了内存错误，系统核心就会崩溃。

有没有一种语言可以控制这样的风险呢？

后来的 James Gosling 在 1995 年开发出的 Java 语言继承了 C 和 C++语言的优点，摒弃了 C++语言里的指针操作、手动管理内存、多继承等诸多复杂而并不实用的功能特性，引入了划时代的 Java 虚拟机（Java Virtual Machine，JVM）。JVM 是一种虚拟的计算机，从结构上看，它与实际的计算机架构相似，JVM 的作用是使得一台实际的机器能够运行 Java 字节码（bytecode）。引用 Java 之父 James Gosling 的话就是

“大部分人大谈特谈 Java 语言，这对于我来说也许听起来很奇怪，但是我无法不去在意。JVM 才是 Java 生态系统的核心啊。我真正关心的是 Java 虚拟机的概念，因为是它把所有的东西都联系在了一起；是它造就了 Java 语言；是它使得事物能在所有的异构平台上得到运行；也还是它使得所有类型的语言能够共存。”

首先，JVM 实现了 Java 的可移植性。另外，JVM 里面实现了一个垃圾收集器（Garbage Collector，GC）来管理内存，GC 对保证系统的可靠性和安全性非常实用有益。同时，JVM 还奠定了一个庞大的语言生态的基础。

Java 是互联网时代当之无愧的最流行的开发语言。经过 20 多年的积累和沉淀，Java 生态拥有了很多优秀的开源社区，如 Apache 和 Spring。有了这些框架，我们可以更加专注业务的实现。

Java 语言也有不好的一面，简单列举如下。

1. 检查异常（Checked Exceptions）

检查异常会在编译时强制执行 try catch 处理，同时还需要进行某种排序处理。检查异常是一个失败的实践，几乎所有的主要 API 提供者都反对可检查异常。Kotlin 中摒弃了检查异常。

2. 基本类型和数组

Java 的这个设计保留了字节码的底层细节，违反了“凡事皆为对象”的原则，如泛型无法包容基本类型就是一个经典的例子。这也使得 Java 的类型系统显得不是那么地简单统一。比较好的方案是，源代码不用直接使用基本类型或者数组，由编译器（或者 JVM）来决定是否可以帮你对其进行优化，而 Kotlin 正是这么做的。

3. 静态变量（Static）

静态方法经常会导致需要显式地定义接口，从而使得 API 更加复杂。一个更好的办法就是采用单例对象，单例对象在大多数情况下的表现与静态对象差不多，但是可以像一个对象一样被传递。Kotlin 中提供了 object 单例对象。

4. 泛型

Java 泛型本身就很复杂，当使用?exends 和?super 等变种句型时就变得尤为复杂，非常容易搞错。这个问题在 *Effective Java* 一书中提出了 PECS（Producer Extends Consumer Super）的建议，Kotlin 直接使用了这个方案。

5. 空指针异常（NPE）

在 Java 中我们不得不写一堆防御代码来避免令人头疼的 NPE。Kotlin 中引入了可空类型与安全调用符、Elvis 操作符等特性来实现空安全，这部分内容将在第 3 章中介绍。

6. 一堆getter/setter单调冗长的样板代码

例如，下面的 Person Bean 类：

```
class Person {  //用 Java 声明一个 Person Bean 类
    Integer id;
    String name;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
```

```
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Person(Integer id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

在 Kotlin 中我们可以使用数据类，代码如下：

```
data class Person(val id: Int, val name: String)
                                //使用 Kotlin 的数据类声明一个 Person Bean 类
```

关于数据类的内容，将在第 4 章中介绍。

7. 不容易传递函数

Java 中没有提供一等函数类型，函数式编程（FP）只能通过使用接口类型以及多态特性“曲线”来实现。Java 会将每一个算法（方法）都放入类中，这种限制会出现这样的“荒唐”事：我们只是想要实现一个函数算法，而这个时候必须还要给出一个类来放置这个方法；同样，如果在其他地方要调用这个方法，必须通过创建该类来实现调用。在 Kotlin 中直接提供了一等函数类型（First-Class Function Type），其跟普通类型一样，函数类型可以作为值来传递，也可以作为返回值。

此外，还有其他的经验教训，上面所述只是其中的一部分。

不可否认的是，C、C++和 Java 语言都是非常优秀的编程语言。但是事物总是不断发展变化的。就像 C++语言是对 C 语言的继承与发展，Java 语言是对 C++语言的继承与改造，而 Kotlin 语言也是对 Java 语言的继承与变革。

1.6 JVM 语言生态

下面是一个来自 Java 官网文档（http://docs.oracle.com/javase/8/docs/）里的一张 Java 技术模块架构图，如图 1-3 所示。

为了在 JVM 上正确运行我们的程序，只需要按照规范生成正确的 class 文件，然后加载到 JVM 中执行文件中指定的操作字节指令码（byte code）即可。

在过去 20 多年的发展历程中（1991—2017 年，如果算上最初的称为 Oak 语言的 Java 前生），Java 语言、JVM、API 库和框架、应用工具和 Web 服务器的速度、稳定性和功能方面在不断发展，已被公认为是开发企业级服务的首选技术栈。

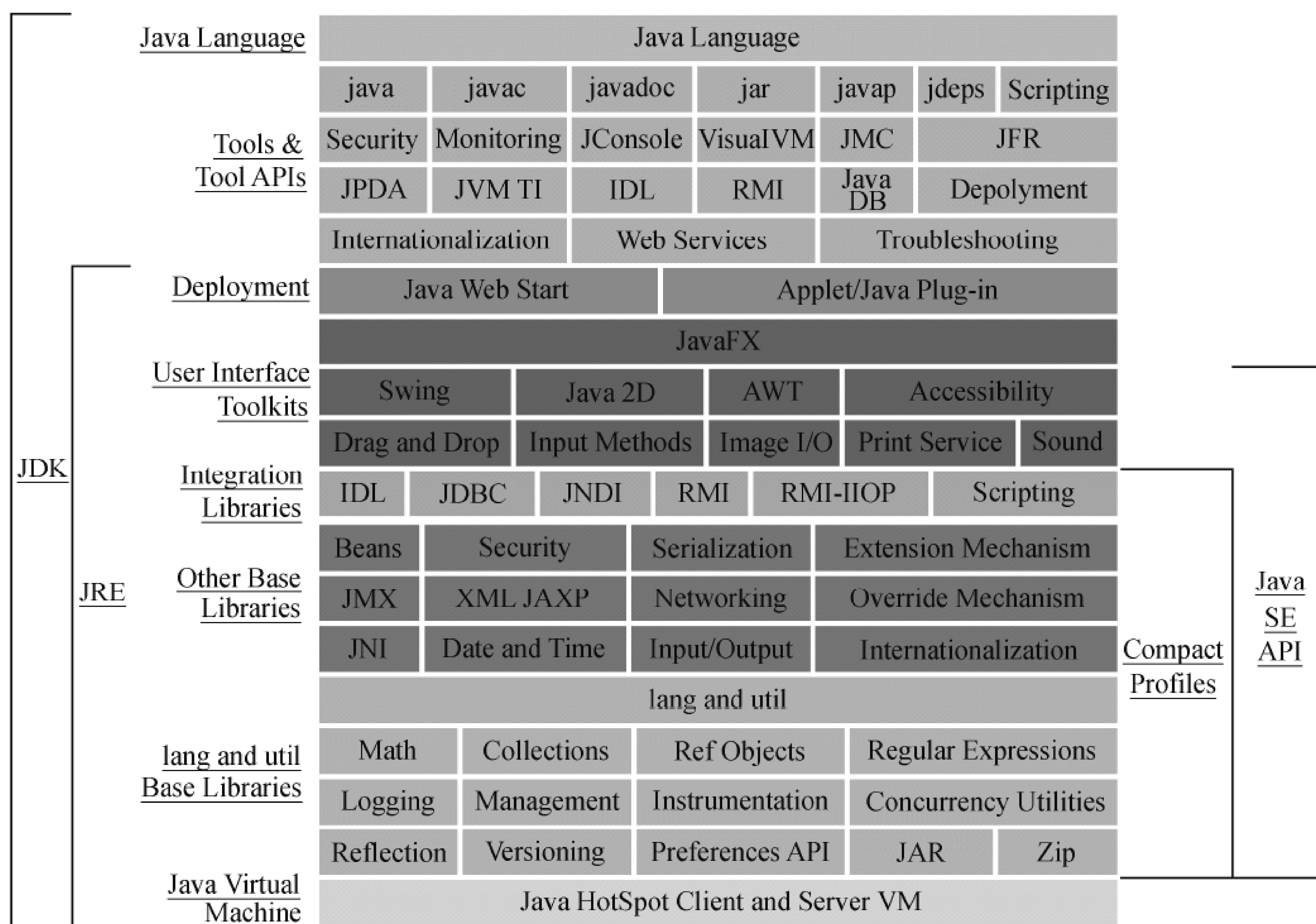


图 1-3　Java 技术模块架构图

JVM 最初是为了支持 Java 编程语言。然而随着时间的推移，越来越多的语言被改编、设计并运行在 JVM 上。除了 Java 语言外，比较知名的 JVM 上的编程语言还有 Groovy、Scala 和 Clojure 等。

JVM 上主流编程语言历史时间轴概览如图 1-4 所示。

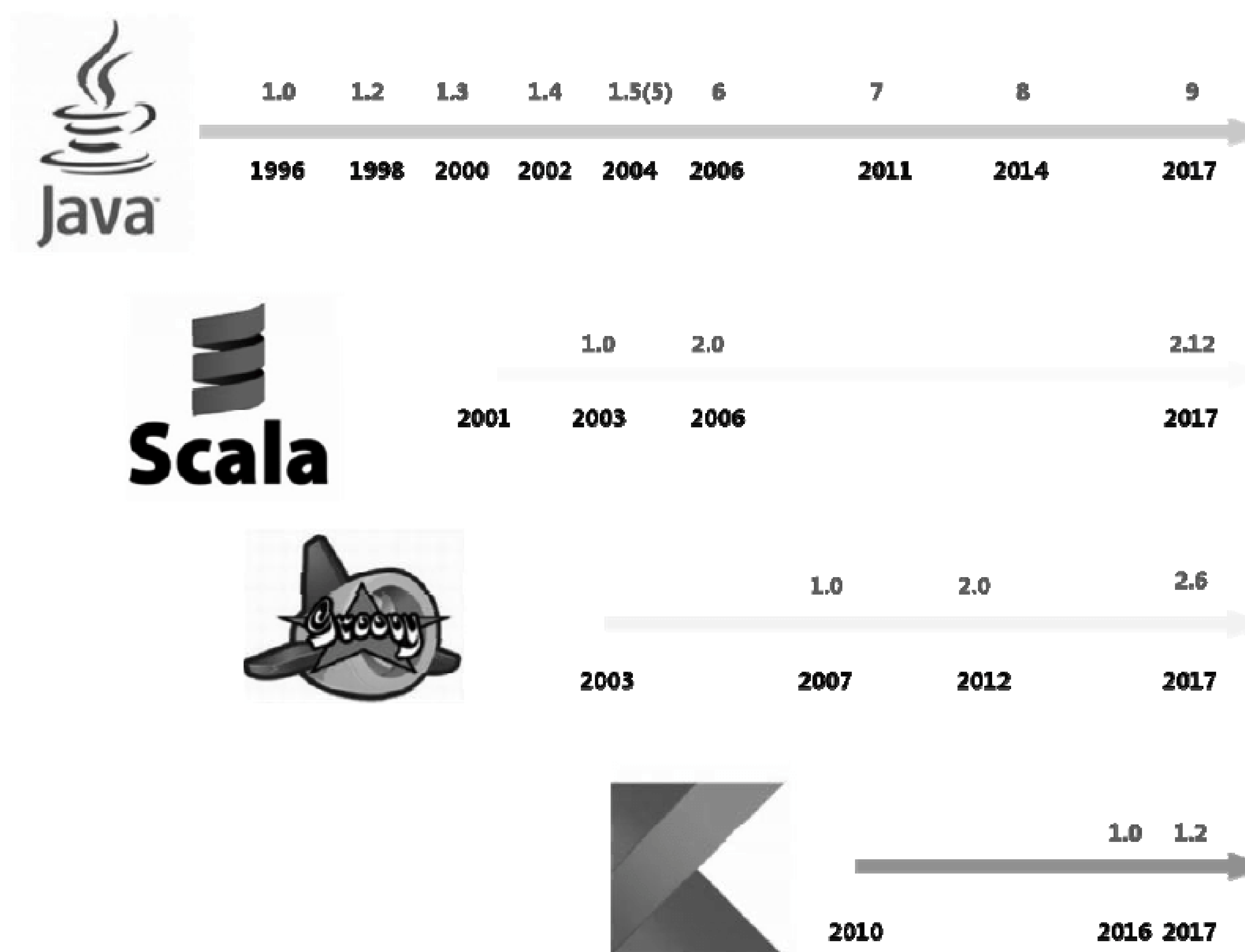


图 1-4　JVM 上主流编程语言历史时间轴概览

说明：计算机中的所有问题，都可以通过向上抽象封装一层来解决。

Java 虚拟机对各个平台而言，实质上是各个平台上的一个可执行程序。例如在 Windows 平台下，Java 虚拟机对于 Windows 而言，就是一个 java.exe 进程而已。

通常情况下，在 JVM 平台上从源代码编译到 JVM 上执行的整体过程如图 1-5 所示。

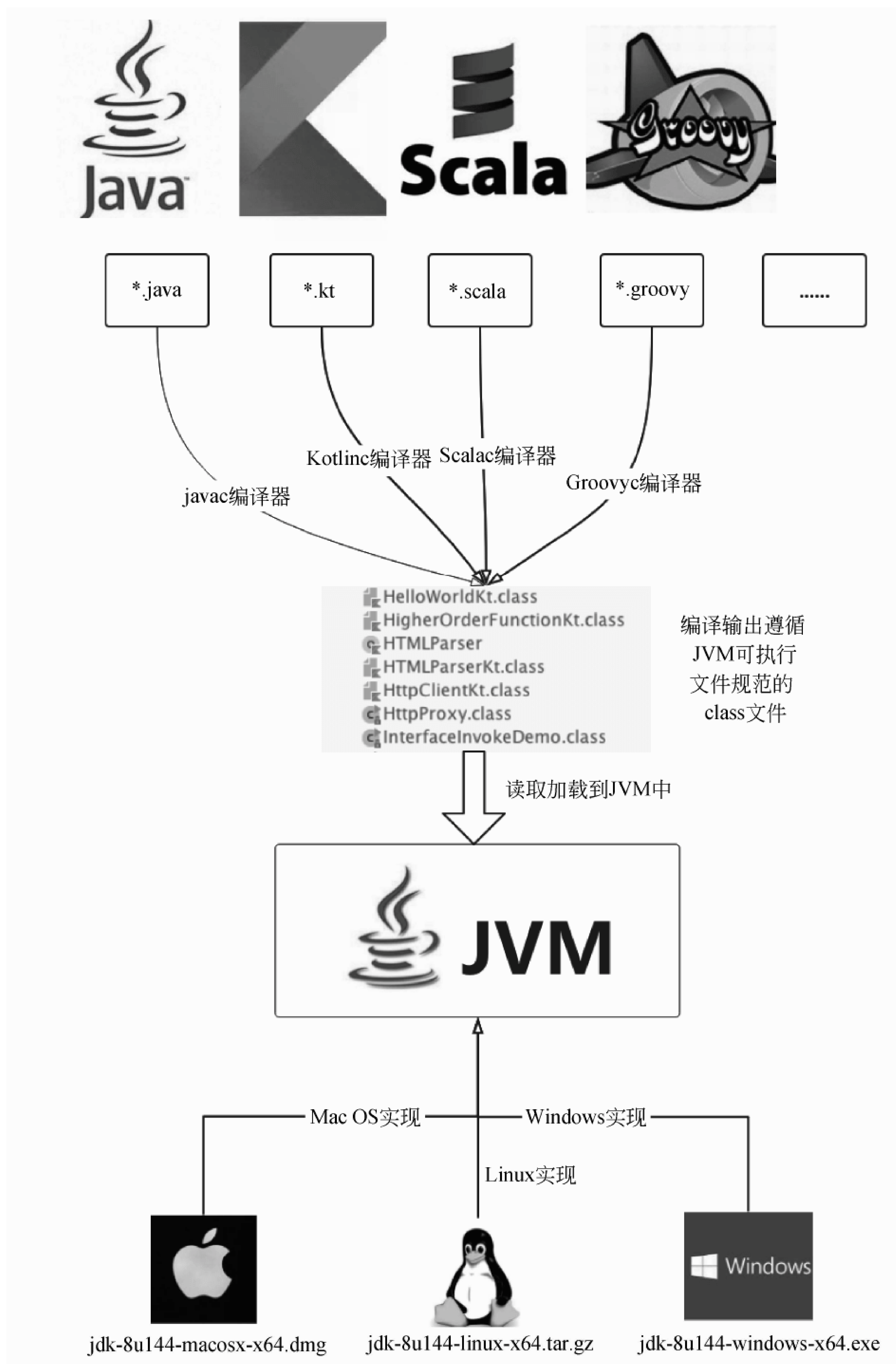


图 1-5　在 JVM 平台上从源代码编译到 JVM 上执行的整体过程

其中，运行在 JVM 上的字节码文件是不依赖于硬件和操作系统的二进制格式的文件。依赖硬件和操作系统的部分，由 JVM 分别在这些平台上来实现。例如，JDK 8 的各个平台的软件版本如图 1-6 所示。

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.89 MB	jdk-8u144-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.83 MB	jdk-8u144-linux-arm64-vfp-hflt.tar.gz
Linux x86	164.65 MB	jdk-8u144-linux-i586.rpm
Linux x86	179.44 MB	jdk-8u144-linux-i586.tar.gz
Linux x64	162.1 MB	jdk-8u144-linux-x64.rpm
Linux x64	176.92 MB	jdk-8u144-linux-x64.tar.gz
Mac OS X	226.6 MB	jdk-8u144-macosx-x64.dmg
Solaris SPARC 64-bit	139.87 MB	jdk-8u144-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.18 MB	jdk-8u144-solaris-sparcv9.tar.gz
Solaris x64	140.51 MB	jdk-8u144-solaris-x64.tar.Z
Solaris x64	96.99 MB	jdk-8u144-solaris-x64.tar.gz
Windows x86	190.94 MB	jdk-8u144-windows-i586.exe
Windows x64	197.78 MB	jdk-8u144-windows-x64.exe

图 1-6　JDK 8 的各个平台的软件版本

我们经常说的 Java 语言是平台无关的，即跨平台的，其实针对从 Java、Scala、Kotlin、Groovy 等的源代码到 JVM 字节码这一层是平台无关的。

但是，真正把 JVM 字节码通过解释器映射到不同平台（操作系统，CPU 硬件架构）上时，JVM 就必须针对各个平台实现一套解释器。只是这一层通过抽象封装，对 Java、Scala、Kotlin、Groovy 程序员而言已经完全透明，无须再做相关的工作而已。

在下一代普遍可接受语言（next mass-appeal language）中，人的因素应该是至关重要的。在功能方面，应该具备如下特性：

- 类 C 的语法（容易被大众程序员所接受，很好用也很熟悉）；
- 静态类型（动态类型过于松散并且性能有限，不适用于大型项目）；
- 遵循面向对象程序设计 OOP 思想，同时支持函数式编程（FP）；
- 反射（从而避免静态类型限制）；
- 属性（getter 和 setter 实在是太让人讨厌了）；
- 高阶函数，Lambda 与闭包；
- Null 判断（提供一个判断变量能否为 null 的方式）；
- 并发协程；
- 模块化；
- 完善的工具支持；
- 可扩展性（语言的设计具备很好的可扩展性）；
- ...

程序语言设计其实堪比艺术品设计，每个开发者的喜好与审美都不同，有自己的风格与特征，所以设计出一门好的编程语言确实不容易。

生命只有一次，所以不要去做一些重复无聊的事情。能交给计算机做的，就尽量交给计算机去做。未来，人工智能将取代大部分的重复手工劳动，将大大解放人类的劳动力，从而使得人类能够花更多的时间和精力去创造，去创新。而人工智能的本质，就是对人类

智能的抽象建模。人类设计的操作系统、浏览器、办公软件、画图设计工具、3D 建模软件、电商系统、金融平台、社交 APP，不就是另一种层次上的人工智能吗？这些东西，背后都是 01 的映射。当然，01 背后是物理层次量子微观的世界了，更加奥妙无穷。

纵览整个计算机的发展史，最重要的思想非“抽象”莫属。一层层的抽象封装了实现的细节，经过计算机技术的不断发展，到了今天的互联网（云计算，大数据，机器智能）时代。

同时，程序员写代码从最初的拿着符号表在纸带上打孔，到使用近似自然语言的高级编程语言来编程（当然背后少不了编译器、解释器，还有的是先通过虚拟机中间字节码这一层，再通过解释器映射到机器码，最后在硬件上进行高、低电平的超高频率的“舞蹈”），以及当今各种库 API、框架、集成开发工具集、智能化的编码提示、代码生成等技术，使得现在的程序员，能够更多地去关注问题本身及逻辑的实现。

从只有少数技术人员会用的命令行的 UNIX、DOS 操作系统，到人性化的 GUI（图形用户界面）操作系统，再到移动互联网时代的智能设备，计算机与互联网越来越多地融入到了人们生活的方方面面。

正如解决数学问题时通常我们会谈“思想”，如反证法、化繁为简等，解决计算机问题也有很多非常出色的“思想”。之所以称为思想，是因为“思想”有拓展性与引导性，可以解决一系列问题。

解决问题的复杂程度直接取决于抽象的种类及质量。将结构、性质不同的底层实现进行封装，向上提供统一的 API 接口，让使用者觉得就是在使用一个统一的资源，或者让使用者觉得自己是在使用一个底层不直接提供而“虚拟”出来的资源。

1.7 本 章 小 结

本章主要介绍了 Kotlin 语言的特性及其编程哲学，以及丰富的工具集的支持。JVM 语言蓬勃发展，而 Kotlin 无疑是当下最具潜力、最具发展前景的语言。第 2 章中我们将学习 Kotlin 语言的语法等基础知识。

第 2 章　Kotlin 语法基础

人与人之间往往通过语言来交流沟通，互相协作。人与计算机之间怎样“交流沟通”呢？答案是编程语言。一种语言包含词、短语、句子、文章等，对应到编程语言中就是关键字、标识符、表达式、源代码文件等。通常，一种编程语言的基本构成要素如图 2-1 所示。

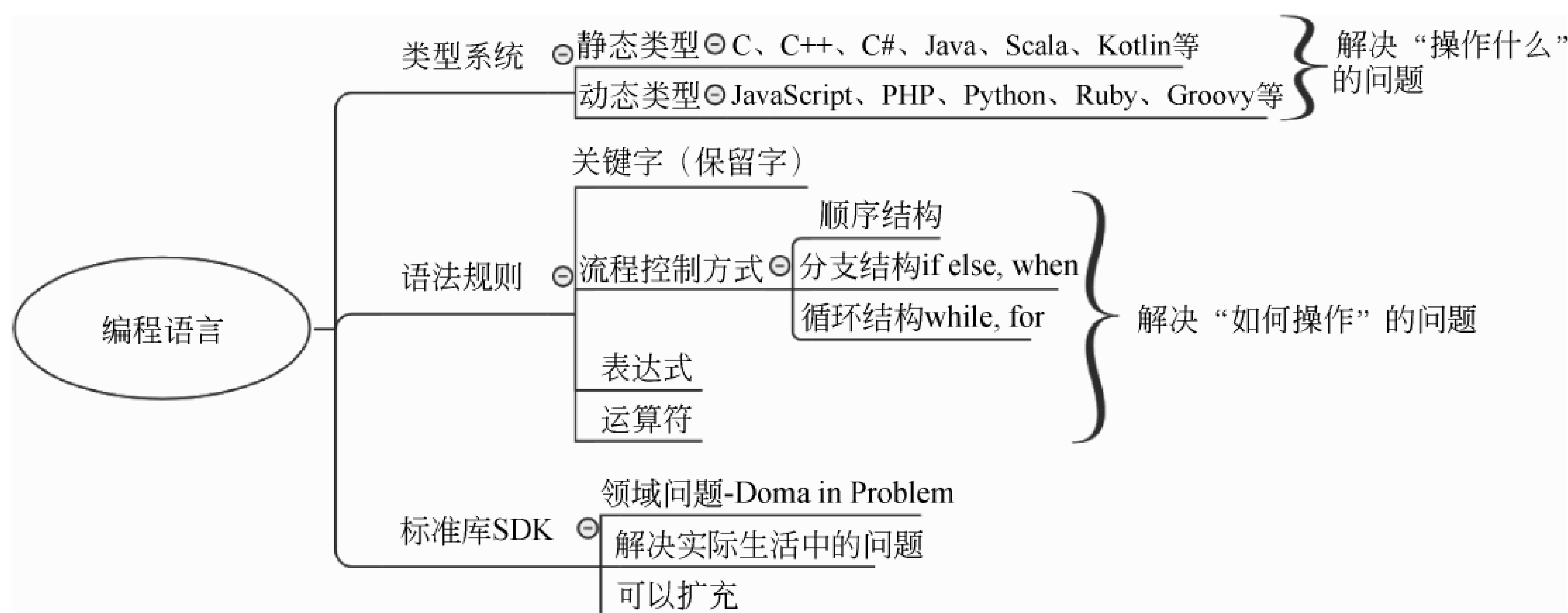


图 2-1　一种编程语言的基本构成要素

本章我们学习 Kotlin 语言的基础语法。

2.1　变量和标识符

变量（数据名称）标识一个对象的地址，我们称之为标识符。而具体存放的数据占用内存的大小和存放的形式则由其类型来决定。

在 Kotlin 中，所有的变量类型都是引用类型。Kotlin 的变量分为 val（不可变的）和 var（可变的）。可以简单理解为：

- val 是只读的，仅能一次赋值，后面就不能被重新赋值；
- var 是可写的，在它生命周期中可以被多次赋值。

例如，使用关键字 val 声明不可变变量，代码如下：

```
>>> val a:Int = 1   //声明一个不可变的 Int 类型的变量 a
>>> a
1
```

另外，还可以省略后面的类型 Int，直接声明如下：

```
>>> val a = 1        //根据值 1 编译器能够自动推断出'Int'类型
>>> a
1
```

用 val 声明的变量不能重新赋值，代码如下：

```
>>> val a = 1
>>> a++              //不可变的对象不能进行重新赋值
error: val cannot be reassigned
a++
^
```

使用 var 声明可变变量，代码如下：

```
>>> var b = 1        //用 var 声明一个可变的变量 b
>>> b = b + 1        //可以对 b 进行重新赋值
>>> b
2
```

只要可以，应尽量在 Kotlin 中首选使用 val 不变值。因为在程序中大部分地方只需要使用不可变的变量，而使用 val 变量可以带来可预测的行为和线程安全等优点。

变量名就是标识符。标识符是由字母、数字、下画线组成的字符序列，不能以数字开头。下面是合法的变量名。

```
>>> val _x = 1                    //下画线开头
>>> val y = 2                     //普通字母
>>> val ip_addr = "127.0.0.1"     //中间带下画线
>>>  x
1
>>> y
2
>>> ip addr
127.0.0.1
```

跟 Java 一样，Kotlin 的变量名区分大小写，命名遵循驼峰式命名法。

2.2　关键字与修饰符

通常情况下，编程语言中都有一些具有特殊意义的标识符是不能用作变量名的，这些具有特殊意义的标识符叫做关键字（又称保留字），编译器需要针对这些关键字进行词法分析，这是编译器对源码进行编译的基础步骤之一。

Kotlin 中的修饰符关键字主要分为：类修饰符、成员修饰符、访问权限修饰符、协变逆变修饰符、函数修饰符、属性修饰符、参数修饰符、具体化类型修饰符等。这些修饰符关键字如表 2-1～表 2-8 所示。

表 2-1　Kotlin中的类修饰符

类 修 饰 符	说　　明
abstract	抽象类
final	不可被继承 final 类
enum	枚举类
open	可继承 open 类
annotation	注解类
sealed	密封类
data	数据类

表 2-2　Kotlin中的成员修饰符

成员修饰符	说　　明
override	重写函数（方法）
open	声明函数可被重写
final	声明函数不可被重写
abstract	声明函数为抽象函数
lateinit	延迟初始化

表 2-3　Kotlin中的访问权限修饰符

访问权限修饰符	说　　明
private	私有，仅当前类可访问
protected	当前类以及继承该类的可访问
public	默认值，对外可访问
internal	整个模块内可访问（模块是指一起编译的一组 Kotlin 源代码文件。例如，一个 Maven 工程，或 Gradle 工程，通过 Ant 任务的一次调用编译的一组文件等）

表 2-4　Kotlin中的协变逆变修饰符

协变逆变修饰符	说　　明
in	消费者类型修饰符，out T 等价于 ? extends T
out	生产者类型修饰符，in T 等价于 ? super T

表 2-5　Kotlin中的函数修饰符

函数修饰符	说　　明
tailrec	尾递归
operator	运算符重载函数
infix	中缀函数。例如，给 Int 定义扩展中缀函数 infix fun Int.shl(x: Int): Int
inline	内联函数
external	外部函数
suspend	挂起协程函数

表 2-6　Kotlin中的属性修饰符

属性修饰符	说　　明
const	常量修饰符

表 2-7 Kotlin中的参数修饰符

<table>
<tr><th>参数修饰符</th><th>说 明</th></tr>
<tr><td>vararg</td><td>变长参数修饰符</td></tr>
<tr><td>noinline</td><td>不内联参数修饰符，有时，只需要将内联函数的部分参数使用内联 Lambda，其他的参数不需要内联，可以使用 noinline 关键字修饰。例如：inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit)</td></tr>
<tr><td>crossinline</td><td>首先，默认内联函数的 Lambda 表达式参数是允许非局部返回的，即：
<pre>
fun outterFun() {
    innerFun {
        return //支持直接返回 outterFun
    }
}
</pre>而使用 crossinline 限制 Lambda 表达式直接非局部 return 返回。
这样做的原因是：
一些内联函数可能调用传给它们的不是直接来自函数体，而是来自另一个执行上下文的 Lambda 表达式参数，如来自局部对象或嵌套函数。在这种情况下，该 Lambda 表达式中也是禁止直接 return 的。为了标识这种情况，该 Lambda 表达式参数需要用 crossinline 修饰符标记</td></tr>
</table>

表 2-8 Kotlin中的具体化类型修饰符

具体化类型修饰符	说 明
reified	具体化类型参数

一个 crossinline 代码实例如下：

```
inline fun f(crossinline body: () -> Unit) { //内联函数 f 的 body 参数是一个 Lambda
    val f = object: Runnable {              //在对象表达式中使用 body 参数
        override fun run() = body()         //参数标记为 crossinline 后，return
                                            操作将不被允许
    }
}
```

除了上面的修饰符关键字之外，还有一些特殊语义的关键字如表 2-9 所示。

表 2-9 Kotlin中的关键字

关 键 字	说 明
package	包声明
as	类型转换
typealias	类型别名
class	声明类
this	当前对象引用
super	父类对象引用
val	声明不可变变量
var	声明可变变量
fun	声明函数
for	for 循环
null	特殊值 null
true	真值

续表

关　键　字	说　　明
false	假值
is	类型判断
throw	抛出异常
return	返回值
break	跳出循环体
continue	继续下一次循环
object	单例类声明
if	逻辑判断 if
else	逻辑判断，结合 if 使用
while	while 循环
do	do 循环
when	条件判断
interface	接口声明
file	文件
field	成员
property	属性
receiver	接收者
param	参数
setparam	设置参数
delegate	委托
import	导入包
where	where 条件
by	委托类或属性
get	get 函数
set	set 函数
constructor	构造函数
init	初始化代码块
try	异常捕获
catch	异常捕获，结合 try 使用
finally	异常最终执行代码块
dynamic	动态的
typeof	类型定义，预留用

这些关键字定义在源码 org.jetbrains.kotlin.lexer.KtTokens.java 中。

2.3　流程控制语句

流程控制语句是编程语言中的核心之一，可分为：

- 分支语句（if、when）
- 循环语句（for、while）
- 跳转语句（return、break、continue、throw）

2.3.1 if 表达式

if…else 语句是控制程序流程的最基本形式，其中 else 是可选的。在 Kotlin 中，if 是一个表达式，即它会返回一个值（跟 Scala 一样）。代码示例如下：

```
package com.easy.kotlin

fun main(args: Array<String>) {
    println(max(1, 2))                //调用 max()函数
}

fun max(a: Int, b: Int): Int {        //声明 max()函数
    //表达式返回值
    val max = if (a > b) a else b     //直接使用 if…else 表达式来实现 max 逻辑
    return max
}
```

另外，if 的分支可以是代码块，最后的表达式作为该块的值：

```
fun max3(a: Int, b: Int): Int {
    val max = if (a > b) {  //这里 {} 中的内容是一个代码块
        print("Max is a")
        a                   //最后的表达式作为该代码块的值
    } else {
        print("Max is b")
        b                   //同上
    }
    return max
}
```

if 作为代码块时，最后一行为其返回值。另外，在 Kotlin 中没有类似 true?1:0 这样的三元表达式。对应的写法是使用 if…else 语句：

```
if(true) 1 else 0         //if…else 实现三元表达式的逻辑
```

if…else 语句规则：if 后的括号不能省略，括号里表达式的值必须是布尔型。

代码反例：

```
>>> if("a") 1             //if 中只能是布尔类型的值，此处是字符串"a"，报错
error: type mismatch: inferred type is String but Boolean was expected
if("a") 1
   ^

>>> if(1) println(1)      //if 中传入 Int 类型，报错
error: the integer literal does not conform to the expected type Boolean
if(1)
   ^
```

如果条件体内只有一条语句需要执行，那么 if 后面的大括号可以省略。良好的编程风格是建议加上大括号。

```
>>> if(true) println(1) else println(0)        //if 后面的大括号可以省略
1
```

```
>>> if(true) { println(1)}  else{ println(0)}//良好的编程风格是建议加上大括号
1
```

编程实例：用 if…else 语句判断某年份是否是闰年。

```
fun isLeapYear(year: Int): Boolean {
    var isLeapYear: Boolean                            //声明一个局部变量
    if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)) {
                                                       //判断是否是闰年
        isLeapYear = true
    } else {
        isLeapYear = false
    }
    return isLeapYear
}

fun main(args: Array<String>) {
    println(isLeapYear(2017))                          //不是闰年
    println(isLeapYear(2020))                          //是闰年
}
```

2.3.2　when 表达式

when 表达式类似于 switch…case 表达式。when 会对所有的分支进行检查直到有一个条件被满足。但相比 switch 而言，when 语句的功能要更加强大、灵活。

Kotlin 的极简语法表达风格，使得我们对分支检查的代码写起来更加简单、直接：

```
fun casesWhen(obj: Any?) {
    when (obj) {                               //when 表达式
        0,1,2,3,4,5,6,7,8,9 -> println("${obj} ===> 这是一个 0-9 之间的数字")
                                               //逗号分隔的序列
        "hello" -> println("${obj} ===> 这个是字符串 hello")      //字符串
        is Char -> println("${obj} ===> 这是一个 Char 类型数据") //is 类型判断
        else -> println("${obj} ===> else 类似于 Java 中的 case-switch 中的
        default")                              //默认路径
    }
}

fun main(args: Array<String>) {
    casesWhen(1)
    casesWhen("hello")
    casesWhen('X')
    casesWhen(null)
}
```

输出如下：

```
1 ===> 这是一个 0-9 之间的数字
hello ===> 这个是字符串 hello
X ===> 这是一个 Char 类型数据
null ===> else 类似于 Java 中的 case-switch 中的 default
```

像 if 语句一样，when 语句的每一个分支也可以是一个代码块，它的值是块中最后的表达式的值。如果其他分支都不满足条件会到 else 分支（类似 default）。

如果我们有很多分支需要用相同的处理方式，则可以把多个分支条件放在一起，用逗号分隔：

```
0,1,2,3,4,5,6,7,8,9 -> println("${obj} ===> 这是一个 0-9 之间的数字")
```

可以用任意表达式（而不只是常量）作为分支条件：

```
fun switch(x: Int) {
    val s = "123"
    when (x) {
        -1, 0 -> print("x == -1 or x == 0")
        1 -> print("x == 1")
        2 -> print("x == 2")
        8 -> print("x is 8")
        parseInt(s) -> println("x is 123")
        else -> {          //注意这个块
            print("x is neither 1 nor 2")
        }
    }
}
```

也可以检测一个值在 in 或者不在!in 一个区间或者集合中：

```
val x = 1
val validNumbers = arrayOf(1, 2, 3)
when (x) {
    in 1..10 -> print("x is in the range")  //是否在范围 1..10
    in validNumbers -> print("x is valid")  //是否在数据 arrayOf(1, 2, 3)中
    !in 10..20 -> print("x is outside the range")   //不在范围 10..20 中
    else -> print("none of the above")       //默认路径
}
```

编程实例：用 when 语句写一个阶乘函数。

```
fun fact(n: Int): Int {
    var result = 1
    when (n) {
        0, 1 -> result = 1           //当 n=0,1 的时候，返回 1
        else -> result = n * fact(n - 1) //除了 0,1 两种情况，递归调用自己 n * fact(n-1)
    }
    return result
}

fact(10) //3628800
```

2.3.3 for 循环

for 循环可以对任何提供迭代器（iterator）的对象进行遍历，语法如下：

```
for (item in collection) {      //for in 循环
    print(item)
}
```

如果想要通过索引遍历一个数组或者一个 list，可以这么做：

```
for (i in array.indices) {      //array.indices 存储了数组 array 的下标序列
```

```
    print(array[i])
}
```

其中，array.indices 持有数组的下标列表。我们也可以使用函数 withIndex()来遍历下标与对应的元素：

```
for ((index, value) in array.withIndex()) {     //带下标 index 来访问数据
    println("the element at $index is $value")
}
```

另外，范围（Ranges）表达式也可用于循环中：

```
for (i in 1..10) {           //等同于 1 <= i && i <= 10
    println(i)
}
```

代码简写如下：

```
(1..10).forEach { print(it) }
```

其中的操作符形式的 1..10 等价于 1.rangeTo(10) 函数调用，由 in 和!in 进行连接。

编程实例：编写一个 Kotlin 程序在屏幕上输出 1！+2！+3！+…+10！的和。

我们使用上面的 fact()函数，代码实现如下：

```
fun sumFact(n: Int): Int { //函数声明
    var sum = 0
    for (i in 1..n) {        //for in 循环
        sum += fact(i)       //sum 累加 fact(i)的值
    }
    return sum
}

sumFact(10)                  //4037913
```

2.3.4　while 循环

while 和 do…while 循环语句的使用方式与 C、Java 语言基本一致。代码示例如下：

```
package com.easy.kotlin

fun main(args: Array<String>) {
    var x = 10
    while (x > 0) {      //当 x 大于 0
        x—
        println(x)
    }

    var y = 10
    do {                 //执行 do
        y = y + 1
        println(y)
    } while (y < 20)     //while 判断条件，y 的作用域包含此处
}
```

2.3.5 break 和 continue

break 和 continue 语句都是用来控制循环结构的，主要用来停止循环（中断跳转），但是二者又有区别，下面分别介绍。

break 语句用于完全结束一个循环，直接跳出循环体，然后执行循环后面的语句。

1. 问题场景：打印数字1～10，只要遇到偶数就结束打印

下面我们使用 for 循环打印 1～10 之间的数字，遇到偶数就使用 break 语句结束循环。代码示例如下：

```
for (i in 1..10) {      //for in 循环
    println(i)
    if (i % 2 == 0) {
        break           //直接跳出整个 for 循环体
    }
}                       //break to here
```

输出如下：

```
1
2
```

continue 语句是只终止本轮循环，但是还会继续下一轮循环。可以简单理解为直接在当前语句处中断，跳转到循环入口，执行下一轮循环。而 break 语句则是完全终止循环，跳转到循环出口。

2. 问题场景：打印数字1～10中的奇数

下面我们使用 for 循环来打印 1～10 中的数字，遇到偶数就使用 continue 语句跳过本轮循环，实现只打印奇数的效果。代码示例如下：

```
for (i in 1..10) {
    if (i % 2 == 0) {
        continue        //执行下一个 i 的值，继续 for 循环
    }
    println(i)
}
```

输出如下：

```
1
3
5
7
9
```

2.3.6 return 返回

在 Java 和 C 语言中，return 语句是很常见的了。虽然在 Scala 和 Groovy 类语言中，函

数的返回值可以不需要显示用 return 语句来指定，但是我们仍然认为使用 return 语句的编码风格更容易阅读理解（尤其是在分支流代码块中）。

在 Kotlin 中，除了表达式的值，有返回值的函数都要求显式使用 return 语句返回其值。代码示例如下：

```
fun sum(a: Int,b: Int): Int{
    return a+b   //这里的 return 不能省略
}

fun max(a: Int, b: Int): Int {
 if (a > b){
 return a        //return 不能省略
} else{
 return b        //return 不能省略
}
```

在 Kotlin 中可以直接使用“=”符号返回一个函数的值，这样的函数称为函数字面量。代码示例如下：

```
>>> fun sum(a: Int,b: Int) = a + b                //(1)
>>> fun max(a: Int, b: Int) = if (a > b) a else b //(2)
>>> sum(1,10)
11
>>> max(1,2)
2
```

代码说明如下：

第（1）处使用表达式声明 sum()函数。

第（2）处 ifelse 表达式返回的是一个值，我们直接用表达式来定义一个 max()函数。

```
>>> val sum=fun(a:Int, b:Int) = a+b               //(3)
>>> sum
(kotlin.Int, kotlin.Int) -> kotlin.Int
>>> sum(1,1)
2
```

第（3）处使用 fun 关键字声明了一个匿名函数，并且直接使用表达式来实现函数。需要注意的是，后面的函数体语句中有没有大括号{}代表的意义完全不同。例如下面的代码：

```
>>> val sumf = fun(a:Int, b:Int) = {a+b}       //(4) 直接使用表达式声明函数
>>> sumf
(kotlin.Int, kotlin.Int) -> () -> kotlin.Int   //(5) sumf 是一个高阶函数类型
>>> sumf(1,1)                   //(6) //调用 sumf(1,1)，返回的是一个函数
() -> kotlin.Int                //(7)
>>> sumf(1,1).invoke()          //(8) //使用 invoke 调用函数
2
>>> sumf(1,1)()                 //(9) 使用括号()调用函数，跟 invoke()函数调用等价
2
```

代码说明如下：

第（4）处带上大括号{}的表达式返回的是一个 Lambda 表达式。

第（5）处{a+b}的类型是从(kotlin.Int, kotlin.Int)到()->kotlin.Int 的映射函数。

第（6）处调用了 sumf()函数。

第（7）处中 sumf(1,1)的返回值是一个函数()->kotlin.Int，而不是具体的数值 2。如何实现真正的函数调用呢？请看第（8）处。

第（8）处使用 invoke()函数实现真正调用函数。

在 Kotlin 中，()操作符对应的是类的重载函数，如 invoke()。我们使用()运算符来调用函数。也就是说第（8）处与第（9）处是等价的写法。

我们也可以使用 fun 关键字加上函数名来声明函数，下面同样是展示带上大括号{}的例子，代码示例如下：

```
>>> fun sumf(a:Int,b:Int) = {a+b} //直接使用表达式声明函数，注意到这里的{}表示
                                   一个 Lambda
>>> sumf(1,1)                     //返回类型是一个函数 () -> kotlin.Int
() -> kotlin.Int
>>> sumf(1,1).invoke()            //使用 invoke 调用函数
2
>>> fun maxf(a:Int, b:Int) = {if(a>b) a else b}
>>> maxf(1,2)               //跟上面的 sumf()函数类似，这里的返回类型也是一个函数
() -> kotlin.Int
>>> maxf(1,2).invoke()      //使用 invoke()函数调用 maxf(1,2)函数
2
```

可以看出，sumf 和 maxf()的返回值都是函数类型：

```
() -> kotlin.Int
```

这点与 Scala 是不同的。在 Scala 中，带不带大括号{}的意义是一样的：

```
scala> def maxf(x:Int, y:Int) = { if(x>y) x else y }
                           //scala 的函数表达式带{}，返回类型依然是 Int
maxf: (x: Int, y: Int)Int

scala> def maxv(x:Int, y:Int) = if(x>y) x else y
                           //scala 不带{}与上面带{}的语法等价
maxv: (x: Int, y: Int)Int

scala> maxf(1,2)           //直接调用 maxf(1,2) 函数，返回值是一个 Int
res4: Int = 2

scala> maxv(1,2)           //同上
res6: Int = 2
```

可以看出，maxf: (x: Int,y:Int)Int 与 maxv: (x: Int, y: Int)Int 的签名是一样的。在这里，Kotlin 与 Scala 在大括号的使用上是完全不同的。调用函数方式是直接调用 invoke()函数 sumf(1,1).invoke()。

Kotlin 中 return 语句会从最近的函数或匿名函数中返回，但是在 Lambda 表达式中遇到 return 语句时，则直接返回最近的外层函数。例如下面两个函数是不同的：

```
val intArray = intArrayOf(1, 2, 3, 4, 5) //声明一个 Int 数组
intArray.forEach {
    if (it == 3) return //在 Lambda 表达式中的 return 直接返回最近的外层函数
    println(it)
}
```

输出如下：

```
1
2
```

遇到 3 时会直接返回（类似于循环体中的 break 语句）。

而我们给 forEach 传入一个匿名函数 fun(a:Int)，这个匿名函数里的 return 语句不会跳出 forEach 循环，有点像 continue 语句的效果：

```
val intArray = intArrayOf(1, 2, 3, 4, 5)
intArray.forEach(fun(a: Int) { //这是一个匿名函数
    if (a == 3) return //从最近的函数中返回，也就是上面的匿名函数 fun(a:Int)，但
                         是循环会继续
     println(a)
})
```

输出如下：

```
1
2
4
5
```

为了显式地指明 return 语句返回的地址，Kotlin 还提供了@Label（标签）来控制返回语句，且看 2.3.7 节的讲解。

2.3.7　标签（label）

在 Kotlin 中任何表达式都可以用标签（label）来标记。标签的格式为标识符后跟@符号，如 abc@、_isOK@都是有效的标签。我们可以用 Label 标签来控制 return、break 或 continue 语句的跳转（jump）行为。

代码示例如下：

```
val intArray = intArrayOf(1, 2, 3, 4, 5)
intArray.forEach here@ {        //here@ 是一个标签
    if (it == 3) return@here    //执行指令跳转到 Lambda 表达式标签 here@处。继续
                                  下一个 it=4 的遍历循环
    println(it)
}
```

输出如下：

```
1
2
4
5
```

我们在 Lambda 表达式开头处添加了标签 here@，可以这么理解：该标签相当于记录了 Lambda 表达式的指令执行入口地址，然后在表达式内部使用 return@here 跳转至 Lambda 表达式中的该地址处。这样代码更加易懂。

另外，也可以使用隐式标签，更加方便。该标签与接收该 Lambda 的函数同名。

代码示例如下：

```
val intArray = intArrayOf(1, 2, 3, 4, 5)
```

```
intArray.forEach {
    if (it == 3) return@forEach      //返回@forEach 处继续下一个循环
    println(it)
}
```

输出如下：

```
1
2
4
5
```

接收该 Lambda 表达式的函数是 forEach，所以我们可以直接使用 return@forEach 跳转到此处执行下一轮循环。

2.3.8 throw 表达式

在 Kotlin 中 throw 是表达式，它的类型是特殊类型 Nothing。该类型没有值，与 C、Java 语言中的 void 意思一样。

```
>>> Nothing::class       //使用::类引用操作符，返回这个 Nothing 类的类型
class java.lang.Void     //Nothing 直接映射到 Java 中的 Void 类型
```

我们在代码中，用 Nothing 来标记无返回的函数：

```
>>> fun fail(msg:String):Nothing{ throw IllegalArgumentException(msg) }
                       //返回类型 Nothing，表示该函数永远不会返回某个值了
>>> fail("XXXX")       //调用 fail()函数，将会直接抛出异常
java.lang.IllegalArgumentException: XXXX
    at Line57.fail(Unknown Source)
```

另外，如果把一个 throw 表达式的值赋给一个变量，需要显式声明类型为 Nothing，代码示例如下：

```
>>> val ex = throw Exception("YYYYYYYY")   //ex 需要显式声明类型为 Nothing，
                                             否则报错
error: 'Nothing' property type needs to be specified explicitly
val ex = throw Exception("YYYYYYYY")
    ^
>>> val ex:Nothing = throw Exception("YYYYYYYY")
                                           //显式声明 ex 的类型为 Nothing
java.lang.Exception: YYYYYYYY
```

另外，因为 ex 变量是 Nothing 类型，没有任何值，所以无法作为参数传给函数。

2.4 操作符与重载

Kotlin 允许我们为自己的类型提供预定义的一组操作符的实现。这些操作符具有固定的符号表示（如“+”或“*”）和固定的优先级。这些操作符的符号定义如下：

```
KtSingleValueToken LBRACKET    = new KtSingleValueToken("LBRACKET", "[");
```

```
KtSingleValueToken RBRACKET    = new KtSingleValueToken("RBRACKET", "]");
KtSingleValueToken LBRACE    = new KtSingleValueToken("LBRACE", "{");
KtSingleValueToken RBRACE      = new KtSingleValueToken("RBRACE", "}");
KtSingleValueToken LPAR        = new KtSingleValueToken("LPAR", "(");
KtSingleValueToken RPAR        = new KtSingleValueToken("RPAR", ")");
KtSingleValueToken DOT         = new KtSingleValueToken("DOT", ".");
KtSingleValueToken PLUSPLUS    = new KtSingleValueToken("PLUSPLUS", "++");
KtSingleValueToken MINUSMINUS  = new KtSingleValueToken("MINUSMINUS",
                                "--");
KtSingleValueToken MUL         = new KtSingleValueToken("MUL", "*");
KtSingleValueToken PLUS        = new KtSingleValueToken("PLUS", "+");
KtSingleValueToken MINUS       = new KtSingleValueToken("MINUS", "-");
KtSingleValueToken EXCL        = new KtSingleValueToken("EXCL", "!");
KtSingleValueToken DIV         = new KtSingleValueToken("DIV", "/");
KtSingleValueToken PERC        = new KtSingleValueToken("PERC", "%");
KtSingleValueToken LT          = new KtSingleValueToken("LT", "<");
KtSingleValueToken GT          = new KtSingleValueToken("GT", ">");
KtSingleValueToken LTEQ        = new KtSingleValueToken("LTEQ", "<=");
KtSingleValueToken GTEQ        = new KtSingleValueToken("GTEQ", ">=");
KtSingleValueToken EQEQEQ      = new KtSingleValueToken("EQEQEQ", "===");
KtSingleValueToken ARROW       = new KtSingleValueToken("ARROW", "->");
KtSingleValueToken DOUBLE_ARROW  = new KtSingleValueToken("DOUBLE_ARROW",
                                  "=>");
KtSingleValueToken EXCLEQEQEQ  = new KtSingleValueToken("EXCLEQEQEQ",
                                "!==");
KtSingleValueToken EQEQ        = new KtSingleValueToken("EQEQ", "==");
KtSingleValueToken EXCLEQ      = new KtSingleValueToken("EXCLEQ", "!=");
KtSingleValueToken EXCLEXCL    = new KtSingleValueToken("EXCLEXCL", "!!");
KtSingleValueToken ANDAND      = new KtSingleValueToken("ANDAND", "&&");
KtSingleValueToken OROR        = new KtSingleValueToken("OROR", "||");
KtSingleValueToken SAFE_ACCESS = new KtSingleValueToken("SAFE_ACCESS", "
                                ?.");
KtSingleValueToken ELVIS       = new KtSingleValueToken("ELVIS", "?:");
KtSingleValueToken QUEST       = new KtSingleValueToken("QUEST", "?");
KtSingleValueToken COLONCOLON  = new KtSingleValueToken("COLONCOLON", "::");
KtSingleValueToken COLON       = new KtSingleValueToken("COLON", ":");
KtSingleValueToken SEMICOLON   = new KtSingleValueToken("SEMICOLON", ";");
KtSingleValueToken DOUBLE_SEMICOLON    = new KtSingleValueToken("DOUBLE_
                                       SEMICOLON", ";;");
KtSingleValueToken RANGE       = new KtSingleValueToken("RANGE", "..");
KtSingleValueToken EQ          = new KtSingleValueToken("EQ", "=");
KtSingleValueToken MULTEQ      = new KtSingleValueToken("MULTEQ", "*=");
KtSingleValueToken DIVEQ       = new KtSingleValueToken("DIVEQ", "/=");
KtSingleValueToken PERCEQ      = new KtSingleValueToken("PERCEQ", "%=");
KtSingleValueToken PLUSEQ      = new KtSingleValueToken("PLUSEQ", "+=");
KtSingleValueToken MINUSEQ     = new KtSingleValueToken("MINUSEQ", "-=");
KtKeywordToken NOT_IN      = KtKeywordToken.keyword("NOT_IN", "!in");
KtKeywordToken NOT_IS      = KtKeywordToken.keyword("NOT_IS", "!is");
KtSingleValueToken HASH        = new KtSingleValueToken("HASH", "#");
KtSingleValueToken AT          = new KtSingleValueToken("AT", "@");
KtSingleValueToken COMMA       = new KtSingleValueToken("COMMA", ",");
```

2.4.1　操作符优先级

Kotlin 中操作符的优先级（Precedence）如表 2-10 所示。

表 2-10　操作符的优先级

优　先　级	标　　题	符　　号
最高	后缀（Postfix）	++, −−, ., ?., ?
	前缀（Prefix）	−, +, ++, −−, !, labelDefinition@
	右手类型运算（Type RHS，right-hand side class type（RHS））	:, as, as?
	乘除取余（Multiplicative）	*, /, %
	加减（Additive ）	+, −
	区间范围（Range）	..
（优先级往下依次递减）	Infix 函数	例如，给 Int 定义扩展 infix fun Int.shl(x: Int): Int {...}，这样调用 1 shl 2，等同于 1.shl(2)
	Elvis 操作符	?:
	命名检查符（Named checks）	in, !in, is, !is
	比较大小（Comparison）	<, >, <=, >=
	相等性判断（Equality)	==, !=, ===, !==
	与（Conjunction）	&&
	或（Disjunction）	||
最低	赋值（Assignment）	=, +=, −=, *=, /=, %=

为实现这些操作符，Kotlin 为二元操作符左侧的类型和一元操作符的参数类型提供了相应的函数或扩展函数。重载操作符的函数需要用 operator 修饰符标记，中缀操作符函数使用 infix 修饰符标记。

2.4.2　一元操作符

一元操作符（unary operation）有前缀操作符、递增和递减操作符等。

1. 前缀操作符

前缀操作符放在操作数的前面，分别如表 2-11 所示。

表 2-11　前缀操作符

表　达　式	翻　译　为
+a	a.unaryPlus()
−a	a.unaryMinus()
!a	a.not()

以下是重载一元减运算符的示例：

```
package com.easy.kotlin

data class Point(val x: Int, val y: Int)              //(1)声明数据类 Point
operator fun Point.unaryMinus() = Point(-x, -y)       //(2) operator 修饰符修
                                                         饰一个重载操作符函数
```

代码说明如下：

第（1）处声明 Point 数据类。

第（2）处使用 operator 关键字实现重载函数 unaryMinus()。

测试代码如下：

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4::class)
class OperatorDemoTest {

    @Test
    fun testPointUnaryMinus() {
        val p = Point(1, 1)
        val np = -p           //直接使用 unaryMinus()重载函数操作符 "-"
        println(np)           //Point(x=-1, y=-1)
    }
}
```

2. 递增和递减操作符

inc()和 dec()函数必须返回一个值，它用于赋值给使用++或--操作的变量。前缀和后缀的表达式的返回值是不同的，具体的取值如表 2-12 所示。

表 2-12　递增和递减操作符

表　达　式	翻　译　为
a++	a.inc()返回值是 a
a--	a.dec()返回值是 a
++a	a.inc()返回值是 a+1
--a	a.dec()返回值是 a-1

2.4.3　二元操作符

Kotlin 中的二元操作符有算术运算符、索引访问操作符、调用操作符、计算并赋值操作符、相等与不等操作符、Elvis 操作符、比较操作符、中缀操作符等。下面我们分别介绍。

1. 算术运算符

Kotlin 的算术运算符有加、减、乘、除、取余、范围操作符等，如表 2-13 所示。

表 2-13　算术运算符

表　达　式	翻　译　为
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)

续表

表　达　式	翻　译　为
a % b	a.rem(b)、a.mod(b)
a..b	a.rangeTo(b)

代码示例如下：

```
>>> val a=10
>>> val b=3
>>> a+b          //加法操作符
13
>>> a-b          //减法操作符
7
>>> a/b          //除法操作符
3
>>> a%b          //取余操作符
1
>>> a..b         //范围操作符
10..3
>>> b..a         //范围操作符
3..10
```

简单的四则运算操作符这里就不多说了。需要注意的是范围运算符 a..b 与 b..a 的区别。

2. 字符串的“+”运算符重载

先用代码举个例子：

```
>>> ""+1         //String 类型重载了加法操作符
1
>>> 1+""         //Int 类型没有重载操作符 plus(other:String)
error: none of the following functions can be called with the arguments supplied:
public final operator fun plus(other: Byte): Int defined in kotlin.Int
public final operator fun plus(other: Double): Double defined in kotlin.Int
public final operator fun plus(other: Float): Float defined in kotlin.Int
public final operator fun plus(other: Int): Int defined in kotlin.Int
public final operator fun plus(other: Long): Long defined in kotlin.Int
public final operator fun plus(other: Short): Int defined in kotlin.Int
1+""
 ^
```

从上面的示例可以看出，在 Kotlin 中 1+""是不允许的（相比 Scala 语言，写这样的 Kotlin 代码显得不太友好），只能显式调用 toString()函数来相加：

```
>>> 1.toString()+""      //先把 Int 类型的 1 转换成 String 再相加 1
```

3. 自定义重载的“+”运算符

下面使用一个计数类 Counter 重载的“+”运算符来增加 index 的计数值。代码示例如下：

```
data class Counter(var index: Int)                      //声明一个计数类

operator fun Counter.plus(increment: Int): Counter {    //重载“+”运算符
   return Counter(index + increment)    //计数器的实现:index 加上 increment
}
```

测试类如下：

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4
@RunWith(JUnit4::class)
class OperatorDemoTest
    @Test
    fun testCounterIndexPlus() {
        val c = Counter(1)    //声明一个 Counter 对象，初始化 index = 1
        val cplus = c + 10    //调用重载的加法运算符
        println(cplus)        //返回 Counter(index=11)
    }
}
```

4. in操作符

in 操作符等价于 contains()函数，如表 2-14 所示。

表 2-14　in操作符

表　达　式	翻　译　为
a in b	b.contains(a)
a !in b	!b.contains(a)

5. 索引访问操作符

索引访问操作符方括号[]转换为调用带有适当数量参数的 get 和 set，如表 2-15 所示。

表 2-15　索引访问操作符

表　达　式	翻　译　为
a[i]	a.get(i)
a[i] = b	a.set(i, b)

6. 调用操作符

小括号调用符()转换为调用 invoke()函数，同样带参数调用也会转换为 invoke()函数中的参数。具体的调用示例如表 2-16 所示。

表 2-16　调用操作符

表　达　式	翻　译　为
a()	a.invoke()
a(i)	a.invoke(i)

7. 计算并赋值操作符

对于赋值操作，例如 a+=b，编译器会试着生成 a=a+b 的代码（这里包含类型检查：a+b 的类型必须是 a 的子类型）。计算并赋值操作符对应的重载函数如表 2-17 所示。

表 2-17　计算并赋值操作符

表　达　式	翻　译　为
a += b	a.plusAssign(b)
a –= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.modAssign(b)

8．相等与不等操作符

Kotlin 中有两种类型的相等性：

- 引用相等===!==（两个引用指向同一对象）；
- 结构相等==!=（使用 equals()判断）。

表 2-18　相等与不等操作符

表　达　式	翻　译　为
a == b	a?.equals(b) ?: (b === null)
a != b	!(a?.equals(b) ?: (b === null))

“==”操作符有些特殊：它被翻译成一个复杂的表达式，用于筛选 null 值。意思是：如果 a 不是 null 则调用 equals(Any?)函数并返回其值；否则（即 a ===null）就计算 b=== null 的值并返回。

当与 null 显式比较时，a==null 会被自动转换为 a===null。

注意：===和!==不可重载。

9．Elvis操作符?:

在 Kotin 中，Elvis 操作符特定是跟 null 进行比较。也就是说

```
y = x?:0                          //使用 Elvis 操作符?:
```

等价于

```
val y = if(x!==null) x else 0   //等价的 if…else 逻辑
```

主要用来作 null 安全性检查。

Elvis 操作符“?:”是一个二元运算符，如果第一个操作数为真，则返回第一个操作数，否则将计算并返回其第二个操作数。它是三元条件运算符的变体，命名灵感来自猫王的发型风格。

Kotlin 中没有这样的三元运算符 true?1:0，取而代之的是 if(true) 1 else 0。而 Elvis 操作符算是精简版的三元运算符。

我们在 Java 中使用的三元运算符的语法通常要重复变量两次，示例如下：

```
String name = "Elvis Presley";
String displayName = (name != null)? name: "Unknown"; //Java 中的三元操作符
```

可以使用 Elvis 操作符取而代之：

```
String name = "Elvis Presley";
String displayName = name?:"Unknown"            //使用 Elvis 操作符
```

可以看出，用 Elvis 操作符“?:”可以把带有默认值的 if…else 结构写得极其简短。用 Elvis 操作符不用检查 null（避免了 NullPointerException），也不用重复变量。

Elvis 操作符的这个功能在 Spring 表达式语言（SpEL）中有提供，在 Kotlin 中当然没有理由不支持这个特性了。

代码示例如下：

```
>>> val x = null
>>> val y = x?:0 //等价的逻辑是：if(x!==null) x else 0，此处 x===null，所以
                   选择 else 分支，返回 0
>>> y
0
>>> val x = false
>>> val y = x?:0 //x 是 false，这个时候 x!==null，所以返回值就是 x 本身 false
>>> y
false
>>> val x = ""
>>> val y = x?:0 //x!==null，返回 x 的值
>>> y

>>> val x = "abc" //x!==null，返回 x 的值
>>> val y = x?:0
>>> y
abc
```

10. 比较操作符

Kotlin 中所有的比较表达式都转换为对 compareTo()函数的调用，这个函数需要返回 Int 值，具体的对应关系如表 2-19 所示。

表 2-19　比较操作符

表　达　式	翻　译　为
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0

11. 用infix函数自定义中缀操作符

我们可以通过自定义 infix 函数来实现中缀操作符。代码示例如下：

```
data class Person(val name: String, val age: Int) //声明 Person 数据类

infix fun Person.grow(years: Int): Person { //声明 Person 类型的中缀操作符函数
    return Person(name, age + years)
}
```

测试代码如下：

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4::class)
class InfixFunctionDemoTest {

    @Test fun testInfixFuntion() {
        val person = Person("Jack", 20)
        println(person.grow(2))  //直接调用函数
        println(person grow 2)   //中缀表达式调用方式
    }
}
```

输出如下：

```
Person(name=Jack, age=22)
Person(name=Jack, age=22)
```

2.5　包　声　明

我们在*.kt 源文件开头声明 package 命名空间。例如，在 PackageDemo.kt 源代码中，按照如下方式声明包：

```
package com.easy.kotlin            //声明包

fun what(){                        //包级函数
    println("This is WHAT ?")
}

fun main(args:Array<String>){    //一个包下面只能有一个 main()函数
    println("Hello,World!")
}

class Motorbike{                   //包里面的类
    fun drive(){
        println("Drive The Motorbike ...")
    }
}
```

Kotlin 中的目录与包的结构无须匹配，源代码文件可以在文件系统中的任意位置。

如果一个测试类 PackageDemoTest 与 PackageDemo 在同一个包下面，我们就不需要单独去导入类和包级函数，可以在代码里直接调用。

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4::class)
class PackageDemoTest {
```

```
    @Test
    fun testWhat() {
        what()              //同一个包命名空间下的函数可以直接调用
    }

    @Test
    fun testDriveMotorbike(){
        val motorbike = Motorbike()
        motorbike.drive()
    }
}
```

其中，what()函数与 PackageDemoTest 类在同一个包命名空间下，因此可以直接调用，不需要导入。Motorbike 类与 PackageDemoTest 类同理分析。

如果不在同一个包下面，我们就需要导入对应的类和函数。例如，我们在 src\test\kotlin 目录下新建一个 package com.easy.kotlin.test，使用 package com.easy.kotlin 下面的类和函数，示例如下：

```
package com.easy.kotlin.test

import com.easy.kotlin.Motorbike    //导入类 Motorbike
import com.easy.kotlin.what         //导入包级函数 what()
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4::class)
class PackageDemoTest {

    @Test
    fun testWhat() {
        what()
    }

    @Test
    fun testDriveMotorbike() {
        val motorbike = Motorbike()
        motorbike.drive()
    }
}
```

Kotlin 会默认导入一些基础包到每个 Kotlin 文件中：

```
kotlin.*
kotlin.annotation.*
kotlin.collections.*
kotlin.comparisons.* （自 1.1 起）
kotlin.io.*
kotlin.ranges.*
kotlin.sequences.*
kotlin.text.*
```

根据目标平台还会导入额外的包。

- JVM 平台上会默认导入下面的包：

```
java.lang.*
kotlin.jvm.*
```

- JS 平台上会默认导入下面的包：

```
kotlin.js.*
```

2.6 本 章 小 结

通过本章的学习，我们可以看到 Kotlin 的语法相当精简，不像 Java 那么啰嗦，所以写起代码来也是相当方便。在后面的章节中，我们将继续体验 Kotlin 语言的魅力。第 3 章将介绍 Kotlin 的类型系统与可空类型。

第 3 章　类型系统与可空类型

与 Java、C 和 C ++语言一样，Kotlin 语言也是“静态类型的编程语言”。通常，编程语言中的类型系统中定义了：

- 如何将数值和表达式归为不同的类型；
- 如何操作这些类型；
- 这些类型之间如何互相作用。

我们在编程语言中使用类型的目的是为了让编译器能够确定类型所关联的对象需要分配多少空间。

类型系统在各种语言之间有非常大的差异，主要的差异表现在编译时期的语法及运行时期的操作实现方式上。在每一种编程语言中，都有一个特定的类型系统。静态类型在编译时期就能可靠地发现类型错误，因此通常能增进最终程序的可靠性。然而，有多少的类型错误发生，以及有多少比例的错误能被静态类型所捕获，仍有争论。

本章简单介绍一下 Kotlin 的类型系统。

3.1　类 型 系 统

定型（typing，又称类型指派）的过程就是赋予一组比特以具体的意义。类型通常和存储器中的数值或对象（如变量）相联系。因为在计算机中，任何数值都是由一组简单的比特位组成的，硬件无法区分存储器地址、脚本、字符、整数及浮点数。类型可以告知程序和程序设计者，应该怎么对待那些比特位。

3.1.1　类型系统的作用

使用类型系统，编译器可以检查无意义的、无效的、类型不匹配等错误代码。这也正是强类型语言能够提供更多的代码安全性保障的原因之一。

另外，静态类型检查还可以提供有用的信息给编译器。与动态类型语言相比，由于有了类型的显式声明，静态类型的语言更加易读好懂。

有了类型，我们还可以更好地做抽象化、模块化的工作。这使得我们可以在较高抽象层次思考并解决问题。例如，Java 中的字符数组 char[]s={'a','b','c'} 和字符串类型 String str="abc"就是最简单、最典型的抽象封装实例。下面分别举例说明。

字符数组代码示例如下：

```
jshell> char[] s = {'a','b','c'}      //字符数组
s ==> char[3] { 'a', 'b', 'c' }
jshell> s[0]                          //访问第一个元素，注意下标是 0
$3 ==> 'a'

jshell> s[1]
$4 ==> 'b'

jshell> s[2]
$5 ==> 'c'
```

字符串代码示例如下：

```
jshell> String str = "abc"            //声明字符串
str ==> "abc"

jshell> str.toCharArray();            //String 类型转换成字符数组
$7 ==> char[3] { 'a', 'b', 'c' }
```

3.1.2　Java 类型系统

Java 类型系统可以简单用下面的图 3-1 来表示。

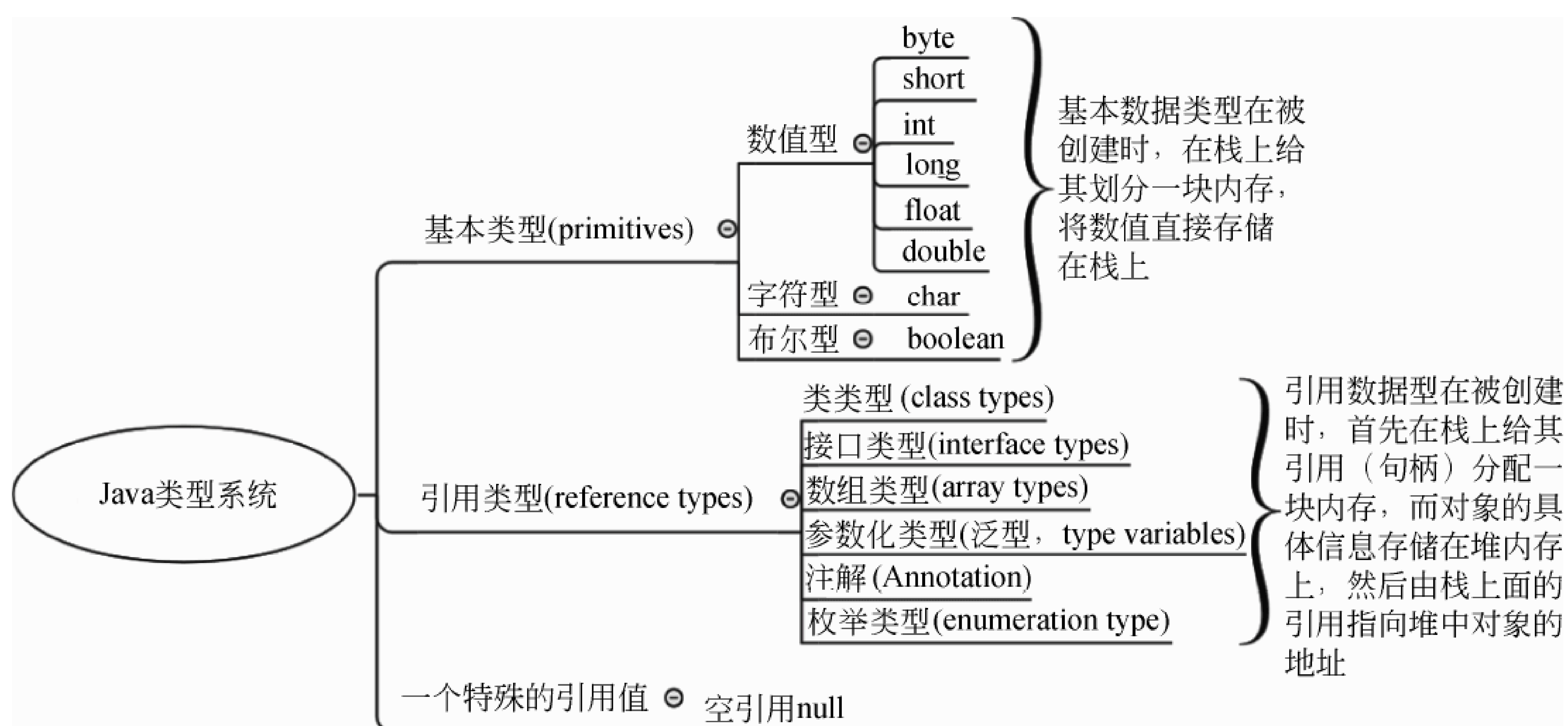


图 3-1　Java 类型系统

关于 Java 中的 null，有很多比较“坑”的地方。例如：

```
int i = null;         //type mismatch : cannot convert from null to int
short s = null;       //type mismatch : cannot convert from null to short
byte b = null:        //type mismatch : cannot convert from null to byte
double d = null;      //type mismatch : cannot convert from null to double
Integer io = null;    //编译
int j = io;           //编译 ok，但运行时报 NullPointerException
```

基本数据类型与引用数据类型在创建时，内存存储方式区别如下：

- ❑ 基本数据类型在被创建时，在栈上给其划分一块内存，将数值直接存储在栈上（性能高）；
- ❑ 引用数据类型在被创建时，首先在栈上给其引用（句柄）分配一块内存，而对象的具体信息存储在堆内存上，然后由栈上面的引用指向堆中对象的地址。

3.1.3　Kotlin 的类型系统

Java 是一个近乎“纯洁”的面向对象编程语言，但是为了编程方便还是引入了基本数据类型。为了能够将这些基本数据类型当成对象操作，Java 为每一个基本数据类型都引入了对应的包装类型（wrapper class），int 的包装类型就是 Integer，从 Java 5 开始引入了自动装箱/拆箱机制，使得二者可以相互转换。Java 为每个原始类型提供了相应的包装类型。

- ❑ 原始类型：boolean，char，byte，short，int，long，float，double。
- ❑ 相应的包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double。

Kotlin 中去掉了原始类型，只有包装类型，编译器在编译代码的时候，会自动优化性能，把对应的包装类型拆箱为原始类型。

Kotlin 系统类型分为可空类型和不可空类型。Kotlin 中引入了可空类型，把有可能为 null 的值单独用可空类型来表示。这样就在可空引用与不可空引用之间划分出一条明确的、显式的“界线”。

Kotlin 类型层次结构如图 3-2 所示。

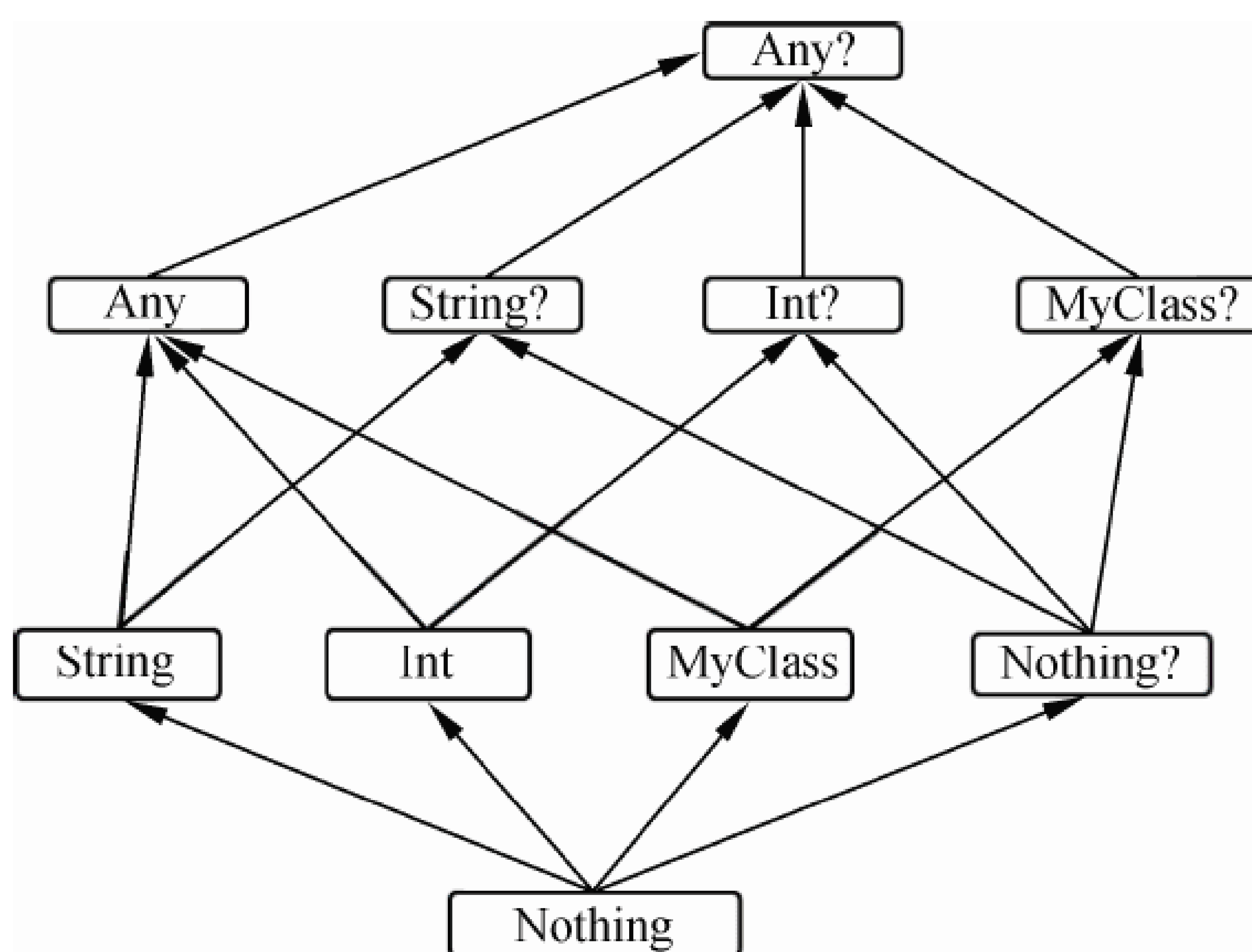


图 3-2　Kotlin 类型层次结构

通过这样显式地使用可空类型，并在编译期作类型检查，大大降低了出现空指针异常的概率。

对于 Kotlin 中的数字类型而言，不可空类型与 Java 中原始的数字类型对应，如表 3-1 所示。

Kotlin 中对应的可空数字类型就相当于 Java 中的装箱数字类型，如表 3-2 所示。

表 3-1　Kotlin中的数字类型与Java中原始的数字类型

Kotlin	Java
Int	int
Long	long
Float	float
Double	double

表 3-2　Kotlin中的可空数字类型与Java中的装箱数字类型

Kotlin	Java
Int?	Integer
Long?	Long
Float?	Float
Double?	Double

在 Java 中，从基本数字类型到引用数字类型的转换过程就是典型的装箱操作，例如 int 转为 Integer。倒过来，从 Integer 转为 int 就是拆箱操作。同理，在 Kotlin 中非空数字类型 Int 到可空数字类型 Int?需要进行装箱操作。同时，非空的 Int 类型会被编译器自动拆箱成基本数据类型 int，存储的时候也会存到栈空间。例如下面的代码，当为 Int 类型的时候，a===b 返回的是 true；而当为 Int?的时候，a===b 返回的是 false。

```
>>> val a: Int = 1000
>>> val b:Int = 1000
>>> a===b        //引用相等
true
>>> a==b         //值相等
true
```

上面返回的都是 true，因为 a、b 它们都是以原始类型存储的，类似于 Java 中的基本数字类型。

```
>>> val a:Int? = 1000
>>> val b:Int? = 1000
>>> a==b
true
>>> a===b //可空类型 Int?等价于 Java 中的 Integer 包装类型，这里 a、b 引用不相等
false
```

可以看出，当 a、b 都为可空类型时，a 与 b 的引用是不等的。“等于”号的简单说明如表 3-3 所示。

表 3-3　Kotlin中的“等于”号说明

“等于”符号	功 能 说 明
=	赋值，在逻辑运算时也有效
==	等于运算，比较的是值，而不是引用
===	完全等于运算，不仅比较值，而且还比较引用，只有两者一致才为真

另外，Java 中的数组也是一个较为特殊的类型。这个类型是 T[]，这个方括号让我们觉得不太“优雅”。Kotlin 中摒弃了这个数组类型声明的语法。Kotlin 简单直接地使用 Array 类型代表数组类型。这个 Array 中定义了 get、set 算子函数，同时有一个 size 属性代表数

组的长度，还有一个返回数组元素的迭代子 Iterator 的函数 iterator()。完整的定义如下：

```
public class Array<T> {
    public inline constructor(size: Int, init: (Int) -> T)
    public operator fun get(index: Int): T
    public operator fun set(index: Int, value: T): Unit
    public val size: Int
    public operator fun iterator(): Iterator<T>
}
```

其中，构造函数我们可以这么用：

```
>>> val square = Array(5, { i -> i * i }) //构造 5 个元素的数组，元素初始值是 i*i
>>> square.forEach(::println)
0
1
4
9
16
```

在编程过程中常用的是 boolean[]、char[]、byte[]、short[]、int[]、long[]、float[]、double[]；Kotlin 直接使用了 8 个新的类型来对应这样的编程场景：

```
BooleanArray
ByteArray
CharArray
DoubleArray
FloatArray
IntArray
LongArray
ShortArray
```

3.2　可空类型

或许 Java 和 Android 开发者早已厌倦了空指针异常（Null Pointer Exception）。因为其在运行时总会在某个意想不到的地方忽然出现，让开发者感到措手不及。

那么为何开发者不能在编译时就提前发现这类空指针异常，并大量修复这些问题呢？现代编程语言正是这么做的。Kotlin 自然也不例外。在 Java 8 中，我们可以使用 Optional 类型来表达可空的类型。

```
package com.easy.kotlin;

import java.util.Optional;
import static java.lang.System.out;

public class Java8OptionalDemo {

    public static void main(String[] args) {
        out.println(strLength(Optional.of("abc")));
        out.println(strLength(Optional.ofNullable(null)));
    }

    static Integer strLength(Optional<String> s) {
        return s.orElse("").length();
```

```
                    //Optional 类型中的 orElse 方法，等价于 Elvis 表达式的逻辑
    }
}
```

运行程序，输出如下：

```
3
0
```

但是这样的代码依然不是那么“优雅”。

针对这方面，Groovy 提供了一种安全的属性/方法访问操作符“?.”：

```
user?.getUsername()?.toUpperCase(); //安全调用符 ?.
```

Swift 也有类似的语法，只作用在 Optional 的类型上。

Kotlin 中使用了 Groovy 里面的安全调用符，并简化了 Optional 类型的使用，直接通过在类型 T 后面加个“?”，就表达了 Optional 的意义。

上面 Java 8 的例子用 Kotlin 来写就显得更加简单、“优雅”了：

```
package com.easy.kotlin

fun main(args: Array<String>) {
    println(strLength(null))
    println(strLength("abc"))
}

fun strLength(s: String?): Int {
    return s?.length ?: 0          //?.是安全调用符，?:是 Elvis 操作符
}
```

其中，我们使用 String?同样表达了 Optional 的意思，相比之下，哪种方式更简单？答案一目了然。

还有 Java 8 的 Optional 提供的 orElse：

```
s.orElse("").length();
```

其在 Kotlin 中是最最常见的 Elvis 运算符了：

```
s?.length ?: 0
```

相比之下，我们还有什么理由继续用 Java 8 的 Optional 呢？

3.3 安全操作符

扔掉 Java 中的一堆 null 的防御式样板代码吧！当我们使用 Java 开发的时候，我们的代码大多是防御性的。如果我们不想遇到 NullPointerException，就需要在使用它之前不停地判断它是否为 null。

Kotlin 正如很多现代编程语言一样是空安全的。因为我们需要通过一个可空类型符号“T?”来明确地指定一个对象类型 T 是否能为空。

我们可以像这样去写：

```
>>> val str: String = null //编译不通过：不可空 String 类型的 str 禁止赋值 null
error: null can not be a value of a non-null type String
val str: String = null
                  ^
```

可以看到，这里不能通过编译，因为 String 类型不能是 null。

一个可以赋值为 null 的 String 类型的正确写法是：String?，代码如下：

```
>>> var nullableStr: String? = null
        //可空类型 String?，nullableStr 可能会导致空指针异常
>>> nullableStrnull
```

我们再来看一下 Kotlin 中关于 null 的一些有趣的运算。null 与 null 是相等的：

```
>>> null==null
true
>>> null!=null
false
```

null 这个值比较特殊，null 不是 Any 类型，例如：

```
>>> null is Any
false
```

但是，null 是 Any?类型，例如：

```
>>> null is Any?
true
```

我们再来看看 null 对应的类型是什么：

```
>>> var a=null
>>> a
null
>>> a=1
error: the integer literal does not conform to the expected type Nothing?
a=1
  ^
```

从报错信息中可以看出，null 的类型是 Nothing?。关于 Nothing?的内容，将会在后面介绍。

3.3.1　安全调用符“?”

我们不能直接使用可空的 nullableStr 来调用其属性或者方法，例如下面的代码直接报错：

```
>>> nullableStr.length
error: only safe (?.) or non-null asserted (!!.) calls are allowed on a
nullable receiver of type String?
nullableStr.length
           ^
```

上面的代码无法编译，nullableStr 可能是 null。我们需要使用安全调用符“?.”来调用：

```
>>> var nullableStr: String? = null
```

```
>>> nullableStr?.length
null
>>> null ableStr = "abc"
>>> nullableStr?.length //安全调用符“?.”
3
```

只有在 nullableStr!=null 时才会去调用其 length 属性。

3.3.2 非空断言“!!”

Kotlin 中提供了断言操作符“!!”，使得可空类型对象可以调用成员方法或者属性（但遇见 null，就会导致空指针异常），代码示例如下：

```
>>> nullableStr = null
>>> nullableStr!!.length             //抛出空指针异常
kotlin.KotlinNullPointerException
```

3.3.3 Elvis 运算符“?:”

使用 Elvis 操作符“?:”来给定一个在 null 情况下的替代值：

```
>>> nullableStr
null
>>> var s= nullableStr?:"NULL" //当 s 是 null 的时候，返回"NULL"字符串
>>> s
NULL
```

3.4 特殊类型

本节我们介绍 Kotlin 中的特殊类型：Unit、Nothing、Any 及其对应的可空类型 Unit?、Nothing?、Any?。

3.4.1 Unit 类型

Kotlin 也是面向表达式的语言。在 Kotlin 中所有控制流语句都是表达式（除了变量赋值、异常等）。

Kotlin 中的 Unit 类型实现了与 Java 中的 void 一样的功能。总的来说，这个 Unit 类型并没有什么特别之处。它的定义如下：

```
package kotlin
public object Unit {                              //Unit 类型是一个 object 对象类型
    override fun toString() ="kotlin.Unit"  //toString()函数返回值
}
```

不同的是，当一个函数没有返回值的时候，我们用 Unit 来表示这个特征，而不是 null。大多数时候，我们并不需要显式地返回 Unit，或者声明一个函数的返回类型为 Unit。

编译器会推断出它。代码示例如下：

```
>>> fun unitExample(){println("Hello,Unit")}
>>> val helloUnit = unitExample()
Hello,Unit
>>> helloUnit                   //函数的返回类型是 Unit
kotlin.Unit
>>> println(helloUnit)
kotlin.Unit
>>> helloUnit is Unit           //判断是否 Unit 类型
true
```

可以看出，变量 helloUnit 的类型是 kotlin.Unit。下面几种写法是等价的：

```
@RunWith(JUnit4::class)
class UnitDemoTest {
    @Test fun testUnitDemo() {
        val ur1 = unitReturn1()
        println(ur1) //kotlin.Unit
        val ur2 = unitReturn2()
        println(ur2) //kotlin.Unit
        val ur3 = unitReturn3()
        println(ur3) //kotlin.Unit
    }

    fun unitReturn1() {                //空函数体，返回类型是 Unit

    }

    fun unitReturn2() {                //显式 return
        return Unit
    }

    fun unitReturn3(): Unit {          //显式声明返回类型 Unit
    }
}
```

跟其他类型一样，Kotlin.Unit 父类型是 Any。如果是一个可空的 Unit?，那么父类型是 Any?。Unit 类型层次结构如图 3-3 所示。

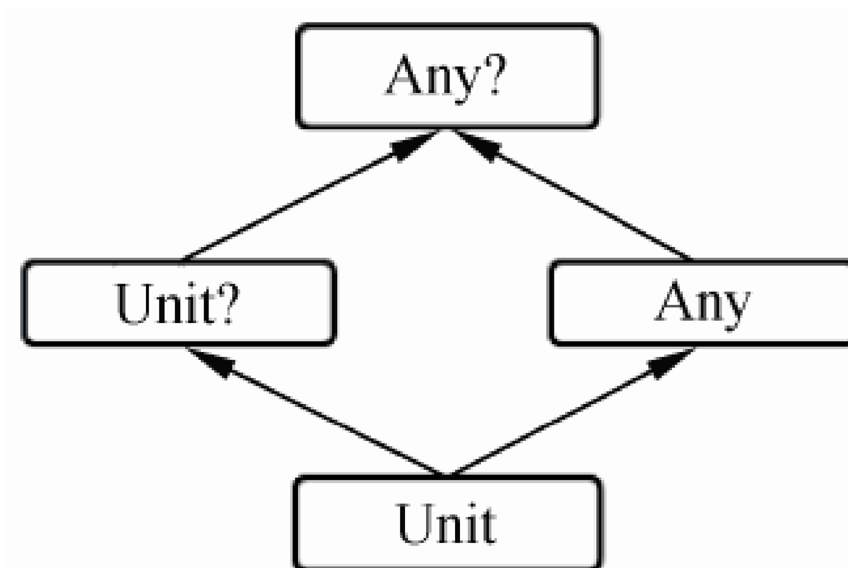


图 3-3　Unit 类型层次结构

3.4.2　Nothing 与 Nothing?类型

在 Java 中，void 不能是变量的类型，也不能作为值打印输出。但是在 Java 中有个包装类 Void 是 void 的自动装箱类型。如果你想让一个方法的返回类型永远是 null 的话，可以把返回类型置为这个大写的 Void 类型。

代码示例如下：

```
public Void voidDemo() {          //声明方法的返回类型是 Void
    System.out.println("Hello,Void");
    return null;                  //这个返回类型 Void 的方法只能返回 null 值
}
```

测试代码如下：

```
@RunWith(JUnit4.class)
public class VoidDemoTest {
    @Test
    public void testVoid() {
        VoidDemo voidDemo = new VoidDemo();
        Void v = voidDemo.voidDemo();    //输出：Hello,Void
        System.out.println(v);           //输出：null
    }
}
```

这个 Void 对应 Kotlin 中的 Nothing?，其唯一可被访问的返回值也是 null。

如 3.13 节中的 Kotlin 类型层次结构图（图 3-2）所示，在 Kotlin 类型层次结构的最底层就是 Nothing 类型，Nothing 类型层次结构如图 3-4 所示。

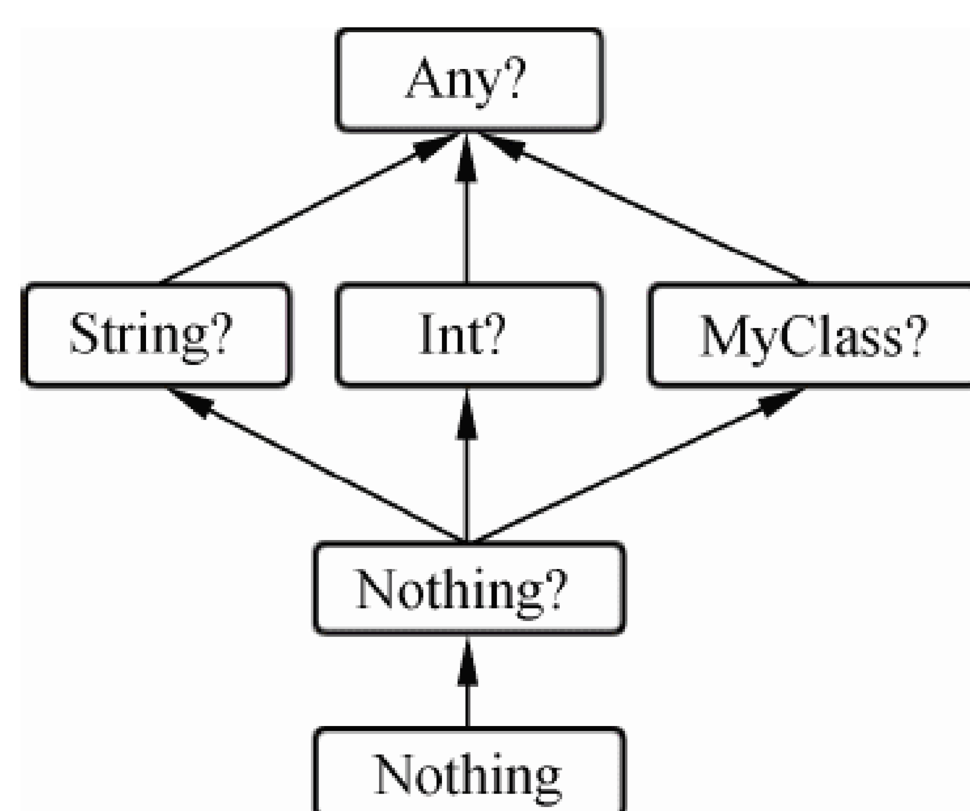


图 3-4　Nothing 类型层次结构

Nothing 类的定义如下：

```
public class Nothing private constructor()
                   //Nothing 的构造函数是 private 的，外界无法创建 Nothing 对象
```

这个 Nothing()不能被实例化：

```
>>> Nothing() is Any    //不能实例化，因为默认的主构造函数是私有的
error: cannot access '<init>': it is private in 'Nothing'
Nothing() is Any
^
```

从上面的代码示例中可以看出 Nothing()不可被访问。如果一个函数的返回值是 Nothing，就意味着这个函数永远不会有返回值。

但是我们可以使用 Nothing 来表达一个从来不存在的返回值。例如 EmptyList 中的 get()函数

```
internal object EmptyList : List<Nothing>, Serializable, RandomAccess {
    override fun get(index: Int): Nothing = throw IndexOutOfBoundsException
    ("Empty list doesn't contain element at index $index.")
                                                //get 函数的返回类型是 Nothing
```

```
    }
}
```

一个空的 List 调用 get()函数，直接抛出了 IndexOutOfBoundsException，这个时候可以使用 Nothing 作为这个 get()函数的返回类型，因为它永远不会返回某个值，而是直接抛出了异常。

再例如 Kotlin 标准库里面的 exitProcess()函数：

```
@file:kotlin.jvm.JvmName("ProcessKt")
@file:kotlin.jvm.JvmVersion
package kotlin.system
@kotlin.internal.InlineOnly
public inline fun exitProcess(status: Int): Nothing {
                                    //exitProcess 函数的返回类型是 Nothing
    System.exit(status)
    throw RuntimeException("System.exit returned normally, while it was
    supposed to halt JVM.")
}
```

注意：Unit 与 Nothing 之间的区别是，Unit 类型表达式计算结果的返回类型是 Unit；Nothing 类型的表达式计算结果是永远不会返回的（与 Java 中的 void 相同）。

Nothing?可以只包含一个值 null。代码示例如下：

```
>>> var nul:Nothing?=null
>>> nul = 1          //Nothing?除了 null 值之外，不能赋其他值
error: the integer literal does not conform to the expected type Nothing?
nul = 1
     ^
>>> nul = true
error: the boolean literal does not conform to the expected type Nothing?
nul = true
     ^
>>> nul = null       //Nothing? 只能赋值为 null 值
>>> nul
null
```

从上面的代码示例中可以看出：Nothing?唯一允许的值是 null，可被用作任何可空类型的空引用。

3.4.3　Any 与 Any?类型

就像 Any 是在非空类型层次结构的根一样，Any?是可空类型层次的根。Any?是 Any 的超集，Any?是 Kotlin 类型层次结构的最顶端。Any 类型层次结构如图 3-5 所示。

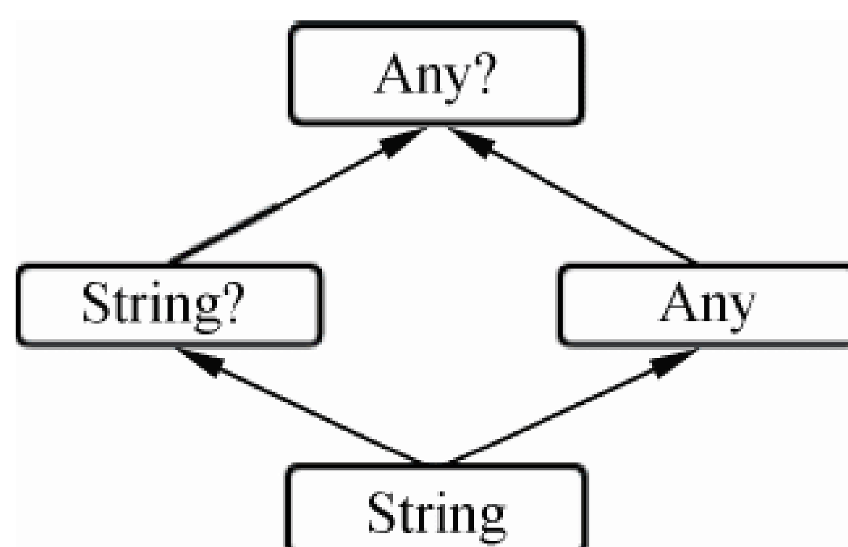


图 3-5　Any 类型层次结构

代码示例：

```
>>> 1 is Any         //Int 类型的 1 是 Any 类型
true
>>> 1 is Any?        //Int 类型的 1 是 Any?类型
true
>>> null is Any      //null 不是 Any 类型
false
>>> null is Any?     //null 是 Any?类型
true
>>> Any() is Any?    //Any()是 Any?类型
true
```

3.5　类型检测与类型转换

Kotlin 在运行时通过使用 is 操作符或其否定形式!is 来检查对象是否符合给定类型。在一般情况下，不需要在 Kotlin 中使用显式转换操作符，因为编译器跟踪不可变值的 is 检查，并在需要时自动插入（安全的）转换。下面分别具体介绍。

3.5.1　is 运算符

is 运算符可以检查对象 A 是否与特定的类型 X 兼容（此对象 A 是 X 类型或者派生于 X 类型），还可以用来检查一个对象（变量）是否属于某数据类型（如 Int、String、Boolean 等）。C#里面也有 is 运算符。

is 运算符类似 Java 中的 instanceof：

```
jshell> "abc" instanceof String    //Java 中的 instanceof 操作符
$10 ==> true
```

在 Kotlin 中，我们可以在运行时通过使用 is 运算符或其否定形式!is，来检查对象是否符合给定类型：

```
>>> "abc" is String
true
>>> "abc" !is String
false

>>> null is Any
false
>>> null is Any?
true
```

代码示例如下：

```
@RunWith(JUnit4::class)
class ISTest {
    @Test fun testIS() {
        val foo = Foo()
        val goo = Goo()
        println(foo is Foo)  //true
```

```
        println(goo is Foo)  //子类 is 父类=true
        println(foo is Goo)  //父类 is 子类=false
        println(goo is Goo)  //true
    }
}

open class Foo
class Goo : Foo()
```

3.5.2　类型自动转换

在 Java 代码中，当我们使用 str instanceof String 来判断其值为 true 的时候，我们想使用 str 变量，还需要显式地强制转换类型：

```
@RunWith(org.junit.runners.JUnit4.class)
public class TypeSystemDemo {
    @org.junit.Test
    public void testVoid() {
        Object str = "abc";
        if (str instanceof String) {
            int len = ((String)str).length();  //显式地强制转换类型为 String
            println(str + " is instanceof String");
            println("Length: " + len);
        } else {
            println(str + " is not instanceof String");
        }
        boolean is = "1" instanceof String;
        println(is);
    }
    void println(Object obj) {
        System.out.println(obj);
    }
}
```

而大多数情况下不需要在 Kotlin 中使用显式转换操作符，因为编译器会跟踪不可变值的 is 检查，并在需要时自动插入（安全的）转换：

```
@Test fun testIS() {
    val len = strlen("abc")
    println(len) //3
    val lens = strlen(1)
    println(lens) //1
}

fun strlen(ani: Any): Int {
    if (ani is String) { //ani 变量的类型如果是 String 类型，编译器会存储它的类型
        return ani.length
                    //这里的 ani 类型已经是 String，可以直接作为 String 类型使用
    } else if (ani is Number) {
        return ani.toString().length
    } else if (ani is Char) {
        return 1
    } else if (ani is Boolean) {
        return 1
    }
    print("Not A String")
    return -1
}
```

3.5.3 as 运算符

as 运算符用于执行引用类型的显式类型转换。如果要转换的类型与指定的类型兼容，转换就会成功进行；如果类型不兼容，使用 as?运算符就会返回 null。

代码示例如下：

```
>>> open class Foo      //父类 Foo
>>> class Goo:Foo()     //子类 Goo
>>> val foo = Foo()
>>> val goo = Goo()
>>> foo as Goo          //父类型不能强制转换为子类型
java.lang.ClassCastException: Line69$Foo cannot be cast to Line71$Goo
>>> foo as? Goo
null
>>> goo as Foo          //子类型可以转换为父类型
Line71$Goo@73dce0e6
```

可以看出，在 Kotlin 中，父类是禁止转换为子类型的。

按照 Liskov 替换原则，父类转换为子类是对 OOP 的严重违反，因为子类除了包含父类所有的方法和属性之外，还可以自定义成员方法与属性，而父类则未必具有和子类同样的成员，所以这种转换是不允许的。

3.6 本 章 小 结

Kotlin 通过引入可空类型，在编译时就大量“清扫了”空指针异常。同时，Kotlin 中还引入了安全调用符“?.”及 Elvis 操作符“?:”，使得我们的代码写起来更加简洁。

Kotlin 的类型系统比 Java 更加简单、一致，Java 中的原始类型与数组类型在 Kotlin 中都统一表现为引用类型。

Kotlin 中还引入了 Unit、Nothing 等特殊类型，使得没有返回值的函数与永远不会返回的函数有了更加规范和一致的签名。

我们可以使用 is 操作符来判断对象实例的类型，使用 as 操作符进行类型的转换。

第 4 章　类与面向对象编程

在前面的章节中，我们学习了 Kotlin 的语言基础知识、类型系统等相关的知识。本章及第 5 章将一起学习 Kotlin 面向对象编程和函数式编程的支持。

本章主要介绍 Kotlin 面向对象编程的知识。

4.1　面向对象编程简史

20 世纪 50 年代后期，在用 Fortran 语言编写大型程序时，由于没有封装机制，那个时候的变量都是“全局变量”，因此会不可避免地经常出现变量名冲突问题。在 ALGOL60 中（1960 年）采用了以 Begin End 为标识的程序块，使块内变量名是局部的，以避免它们与程序中块外的同名变量相冲突。在编程语言中首次提供了封装（保护）机制。此后，程序块结构广泛用于 Pascal、Ada、C 语言等高级语言中。

20 世纪 60 年代中后期，在 ALGOL 基础上研制开发了 Simula 语言，它在 ALGOL 块结构概念的基础上提出了对象的概念并使用了类，也支持类继承。其后的发展简史如图 4-1 所示。

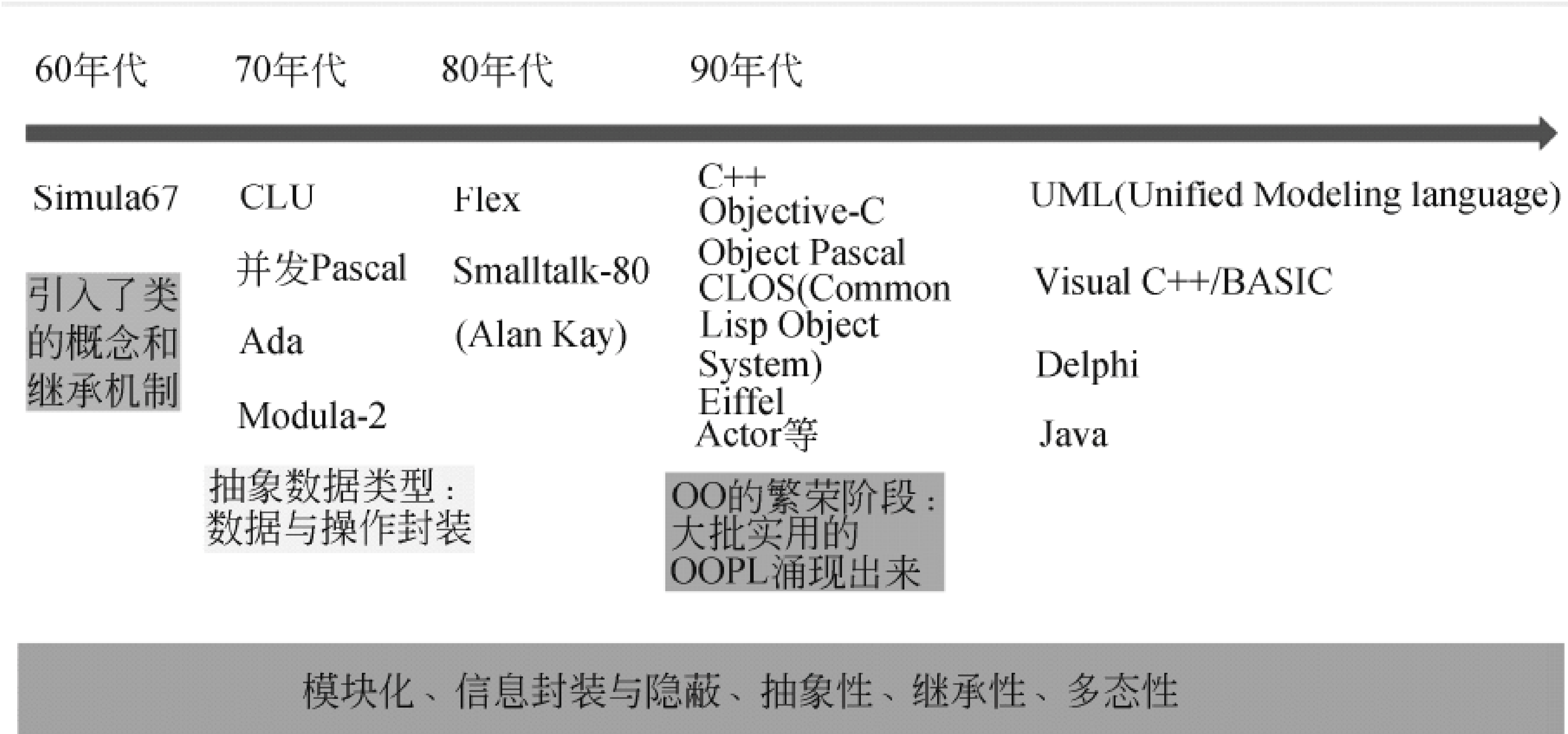


图 4-1　面向对象编程发展简史

阿伦·凯（Alan Kay）是 Smalltalk（1970 年到 1980 年）面向对象编程语言的发明人之一，也是面向对象编程思想的创始人之一，同时，他还是笔记本电脑最早的构想者和现代 Windows GUI 的建筑师。最早提出 PC 概念和互联网的也是阿伦·凯，所以人们都尊称他为“预言大师”。他是当今 IT 界屈指可数的技术天才级人物。

面向对象编程思想主要是复用性和灵活性（弹性）。复用性是面向对象编程的一个主要机制。灵活性主要是应对变化的特性，因为客户的需求是不断改变的，怎样适应客户需求的变化，是软件设计灵活性或者说是弹性的问题。

Java 是一种面向对象编程语言，它基于 Smalltalk 语言，作为 OOP 语言，它具有以下 5 个基本特性。

- 万物皆对象，每一个对象都会存储数据，并且可以对自身执行操作。因此，每一个对象包含两部分：成员变量和成员方法。在成员方法中可以改变成员变量的值。
- 程序是对象的集合，他们通过发送消息来告知彼此所要做的事情，也就是调用相应的成员函数。
- 每一个对象都有自己的由其他对象所构成的存储，也就是说在创建新对象的时候可以在成员变量中使用已存在的对象。
- 每个对象都有其类型，每个对象都是某个类的一个实例，每一个类区别于其他类的特性就是可以向它发送什么类型的消息，也就是它定义了哪些成员函数。
- 某一个特定类型的所有对象都可以接受同样的消息。另一种对对象的描述为：对象具有状态（数据，成员变量）、行为（操作，成员方法）和标识（成员名，内存地址）。

面向对象语言其实是对现实生活中的实物的抽象。

每个对象能够接受的请求（消息）由对象的接口所定义，而在程序中必须由满足这些请求的代码，这段代码称为该接口的实现。当向某个对象发送消息（请求）时，这个对象便知道该消息的目的（该方法的实现已定义），然后执行相应的代码。

我们经常说一些代码片段是优雅的或美观的，实际上是说它们更容易被人类有限的思维所理解。对于程序的复合而言，好的代码是它的“表面积”要比“体积”增长的慢。

代码块的“表面积”是我们复合代码块时所需要的信息（接口 API 协议定义）。代码块的“体积”就是接口内部的实现逻辑（API 背后的实现代码）。

在面向对象编程中，一个理想的对象应该是只暴露它的抽象接口（纯表面，无“体积”），其方法则扮演“箭头”的角色。如果为了理解一个对象如何与其他对象进行复合，而不得不深入挖掘对象的实现之时，你所用的编程范式的优势就荡然无存了。

面向对象编程是一种编程思想，相比于早期的结构化程序设计，其抽象层次更高，思考解决问题的方式也更加贴近人类的思维方式。现代编程语言基本都支持面向对象编程范式。

计算机领域中的所有问题，都可以通过向上一层进行抽象封装来解决。这里封装的本质是概念其实就是“映射”。从面向过程到面向对象，再到设计模式，架构设计，面向服务，Sass、Pass 和 Iass 等思想，各种软件理论思想五花八门，但万变不离其宗：

你要解决一个什么样的问题？

你的问题是哪个领域的？

你的模型（数据结构）是什么？

你的算法是什么？

你对这个世界的本质认知是怎样的？

你的业务领域的逻辑问题、流程是什么？

……

我对 OO 编程的目标从来就不是复用。相反，对我来说，对象提供了一种处理复杂性

的方式。这个问题可以追溯到亚里士多德提出的：您把这个世界视为过程还是对象？在 OO 兴起运动之前，编程以过程为中心，如结构化设计方法。然而，系统已经到达了超越其处理能力的复杂性极点。有了对象，我们能够通过提升抽象级别来构建更大、更复杂的系统，我认为，这才是面向对象编程运动的真正胜利。

面向对象编程以现实世界中的事物（对象）为中心来思考问题，认识问题，并根据这些事物的本质特征，把它们抽象表示为系统中的类。其核心思想可以用图 4-2 进行简要说明。

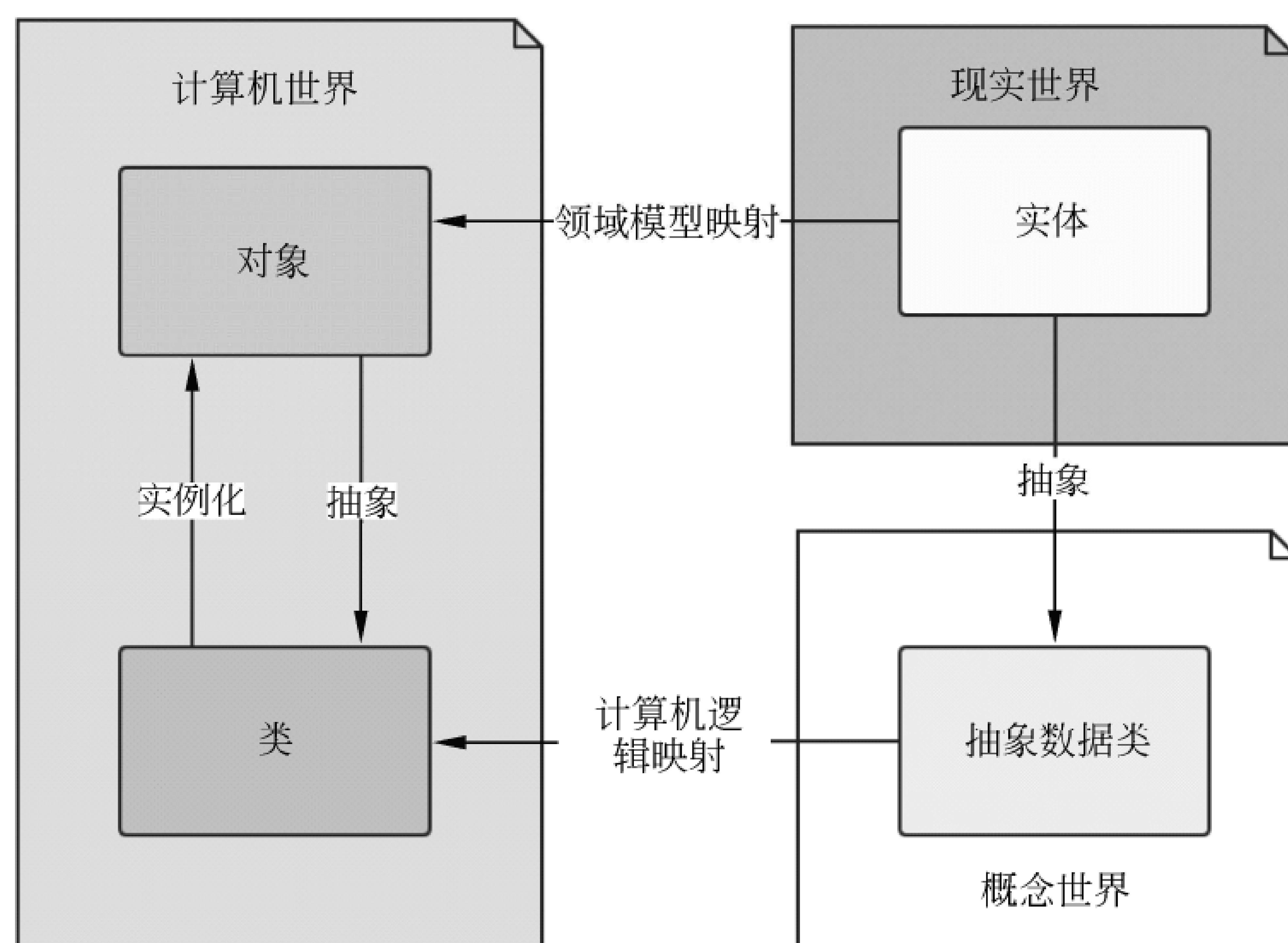


图 4-2　面向对象编程的核心思想

面向对象编程基于类编程，更加贴近人类解决问题的习惯方法，让软件世界更像现实世界。面向对象编程通过抽象出关键的问题域来分解系统。对象不仅能表示具体的事物，还能表示抽象的规则、计划或事件。关于面向对象编程的基本概念如图 4-3 所示。

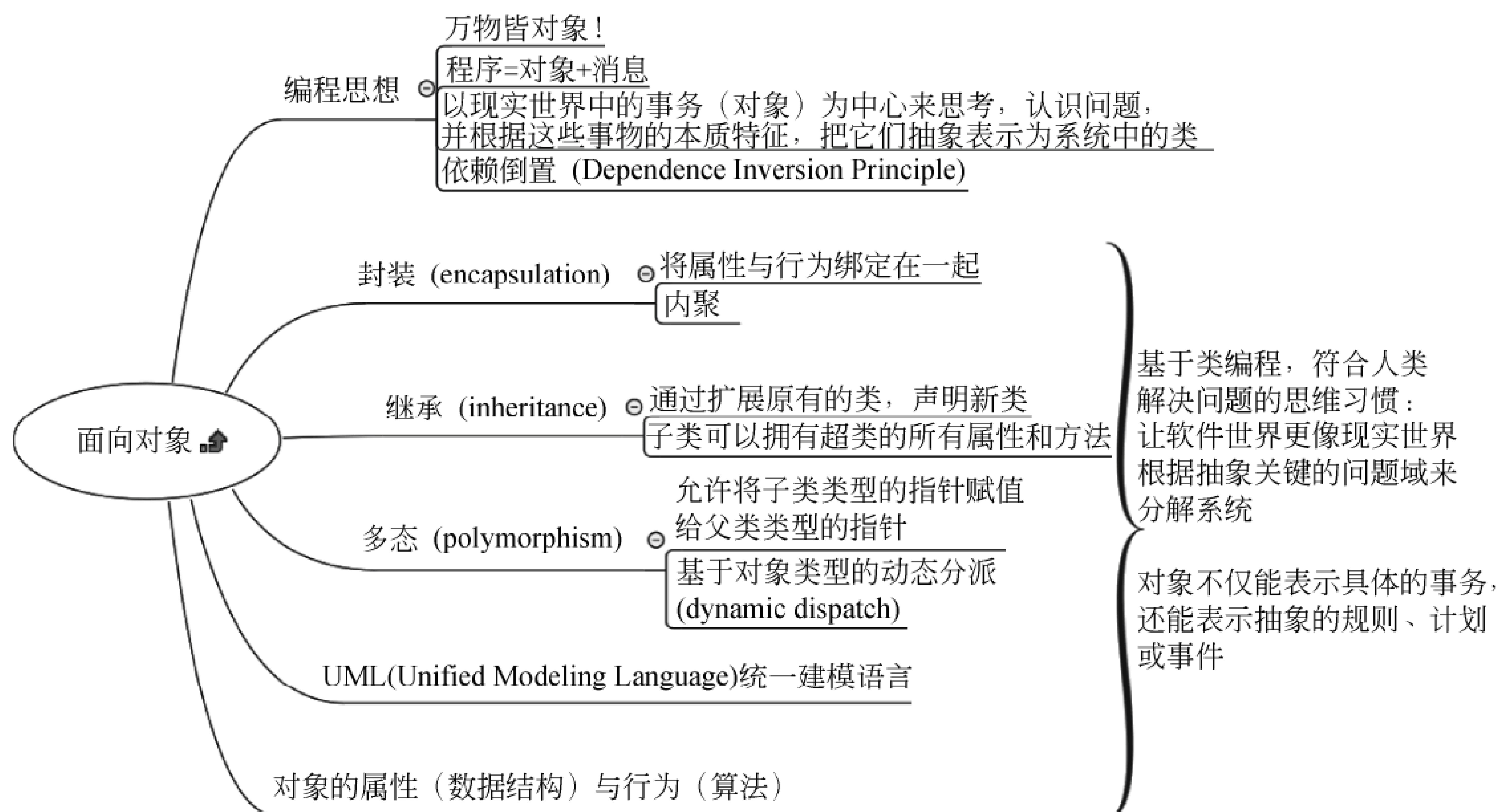


图 4-3　面向对象编程的基本概念

4.2 声 明 类

本节介绍 Kotlin 中的类和构造函数的相关内容。主要包括类的声明和使用，以及构造函数和次级构造函数的编写。

4.2.1 空类

使用 class 关键字声明类。我们可以声明一个什么都不干的类：

```
class AnEmptyClass

fun main(args: Array<String>) {
    val anEmptyClass = AnEmptyClass()       //Kotlin 中不需要使用 new
    println(anEmptyClass)
    println(anEmptyClass is AnEmptyClass)   //对象实例是 AnEmptyClass 类型
    println(anEmptyClass::class)
}
```

输出如下：

```
com.easy.kotlin.AnEmptyClass@2626b418
true
class com.easy.kotlin.AnEmptyClass (Kotlin reflection is not available)
```

4.2.2 声明类和构造函数

在 Kotlin 中，我们可以在声明类的时候同时声明构造函数，语法格式是在类的后面使用括号包含构造函数的参数列表

```
class Person(var name: String, var age: Int, var sex: String) {
                                      //声明类和构造函数
    override fun toString(): String {  //override 关键字，重写 toString()
        return "Person(name='$name', age=$age, sex='$sex')"
    }
}
```

使用这样简洁的语法，可以通过主构造器来定义属性并初始化属性值（这里的属性值可以是 var 或 val）。在代码中可以这样使用 Person 类：

```
val person = Person("Jack", 29, "M")
println("person = ${person}")
```

输出如下：

```
person = Person(name='Jack', age=29, sex='M')
```

另外，也可以先声明属性，等构造实例对象的时候再去初始化属性值，那么 Person 类可以进行如下声明：

```
class Person1 {
```

```
    lateinit var name: String   //lateinit 关键字表示该属性延迟初始化
    var age: Int = 0            //lateinit 关键字不能修饰 primitive 类型
    lateinit var sex: String
    override fun toString(): String {
        return "Person1(name='$name', age=$age, sex='$sex')"
    }
}
```

我们可以在代码中这样创建 Person1 的实例对象：

```
val person1 = Person1()     //声明对象
person1.name = "Jack"       //设置属性值
person1.age = 29
person1.sex = "M"
println("person1 = ${person1}")
```

输出如下：

```
person1 = Person1(name='Jack', age=29, sex='M')
```

如果我们想声明一个具有多种构造方式的类，可以使用 constructor 关键字声明构造函数，示例代码如下：

```
class Person2() {                               //无参的主构造函数
    lateinit var name: String
    var age: Int = 0
    lateinit var sex: String

    constructor(name: String) : this() {       //次级构造函数，this 关键字指向当
                                                 前类对象实例
        this.name = name
    }

    constructor(name: String, age: Int) : this(name) { //次级构造函数
        this.name = name
        this.age = age
    }

    constructor(name: String, age: Int, sex: String) : this(name, age) {
                                                //次级构造函数
        this.name = name
        this.age = age
        this.sex = sex
    }

    override fun toString(): String {
        return "Person1(name='$name', age=$age, sex='$sex')"
    }
}
```

上面的写法总体来看也有一些样板代码，其实在 IDEA 中，上面的代码只需要下面 3 行代码即可替换，剩下的就交给 IDEA 自动生成了。

```
class Person2 {
    lateinit var name: String
    var age: Int = 0
    lateinit var sex: String
}
```

在 IDEA 中自动生成构造函数的操作步骤如下：

（1）在当前类中右击，在弹出的快捷菜单中选择 Generate 命令（在 Mac 上的快捷键是 Command+N），如图 4-4 所示。

（2）之后，弹出生成次级构造函数对话框，在其中选择 Secondary Constructor 命令，如图 4-5 所示。

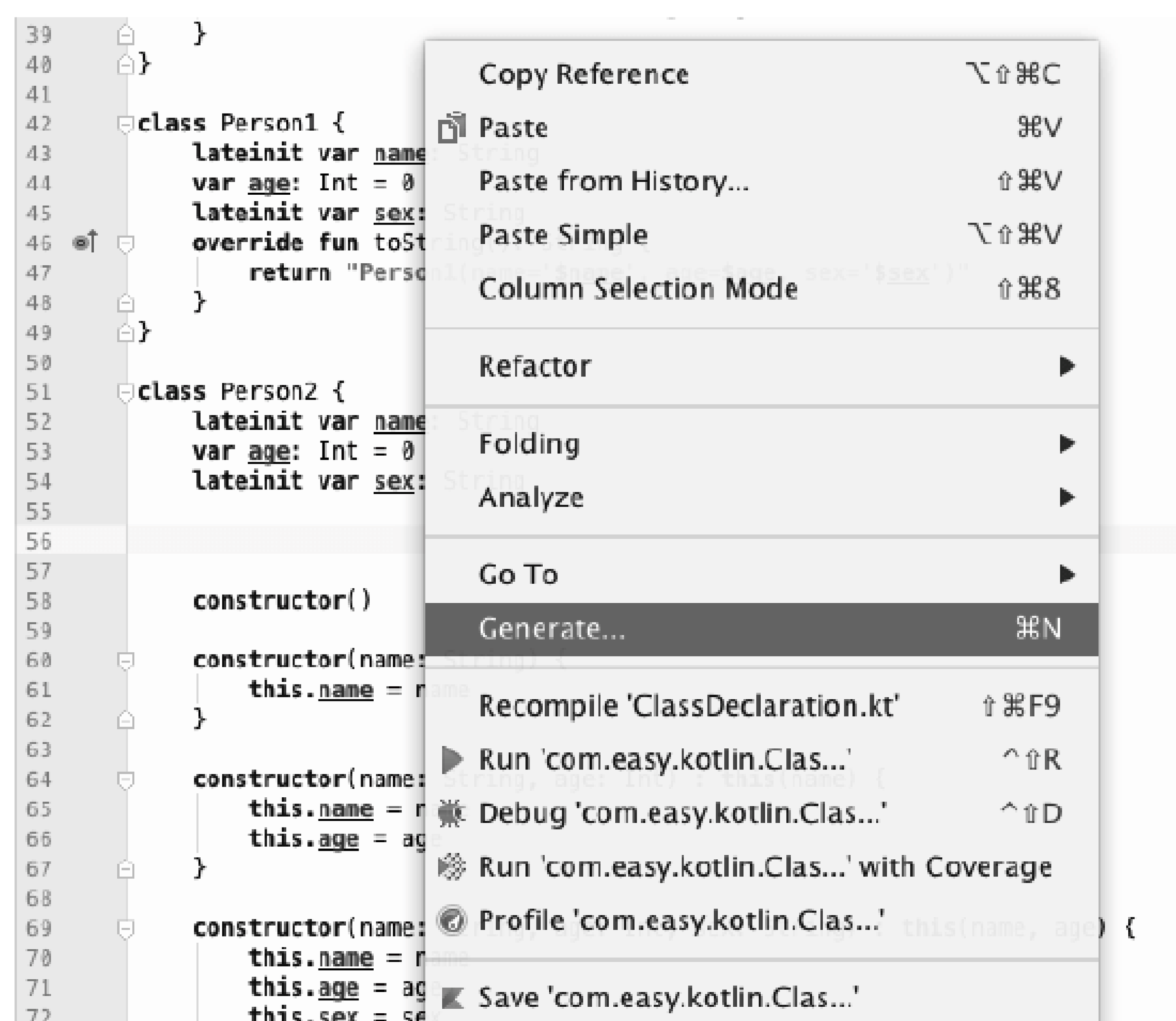


图 4-4　选择 Generate 命令



图 4-5　生成次级构造函数

（3）选择构造函数的参数，如图 4-6 所示。



图 4-6　选择构造函数的成员属性

选中相应的属性，单击 OK 按钮即可生成构造函数。一个属性都不选，生成

```
constructor()
```

选择一个 name 属性，生成带 name 参数的构造函数：

```
constructor(name: String) {
    this.name = name
}
```

选择 name、age 属性，生成这两个参数的构造函数：

```
constructor(name: String, age: Int) : this(name) {
    this.name = name
    this.age = age
}
```

3 个属性都选择，生成这 3 个参数的构造函数：

```
constructor(name: String, age: Int, sex: String) : this(name, age) {
    this.name = name
    this.age = age
    this.sex = sex
}
```

最后，我们可以在代码中这样创建 Person2 的实例对象：

```
val person21 = Person2()
person21.name = "Jack"
person21.age = 29
person21.sex = "M"
println("person21 = ${person21}")

val person22 = Person2("Jack", 29)
person22.sex = "M"
println("person22 = ${person22}")

val person23 = Person2("Jack", 29, "M")
println("person23 = ${person23}")
```

实际上，我们在编程实践中用到最多的构造函数还是这个：

```
class Person(var name: String, var age: Int, var sex: String)
```

当需要通过比较复杂的逻辑来构建一个对象的时候，可采用构建者（Builder）模式来实现。

4.3　抽象类与接口

抽象类表示“is-a”的关系，而接口所代表的是“has-a”的关系。

抽象类用来表征问题领域的抽象概念。所有编程语言都提供抽象机制。机器语言是对机器的模仿抽象，汇编语言是对机器语言的高层次抽象，高级语言（Fortran、C、BASIC 等）是对汇编的高层次抽象。而我们这里所说的面向对象编程语言是对过程函数的高层次封装。这个过程如图 4-7 所示。



图 4-7　编程语言的逐步高层次封装

抽象类和接口是 Kotlin 语言中两种不同的抽象概念，它们的存在对多态提供了非常好的支持。这个机制与 Java 相同。

4.3.1　抽象类与抽象成员

抽象是相对于具象而言的。例如，设计一个图形编辑软件，问题领域中存在着长方形（Rectangle）、圆形（Circle）、三角形（Triangle）等一些具体概念，它们是具象。但是它们又都属于形状（Shape）这个抽象的概念。它们的关系如图 4-8 所示。

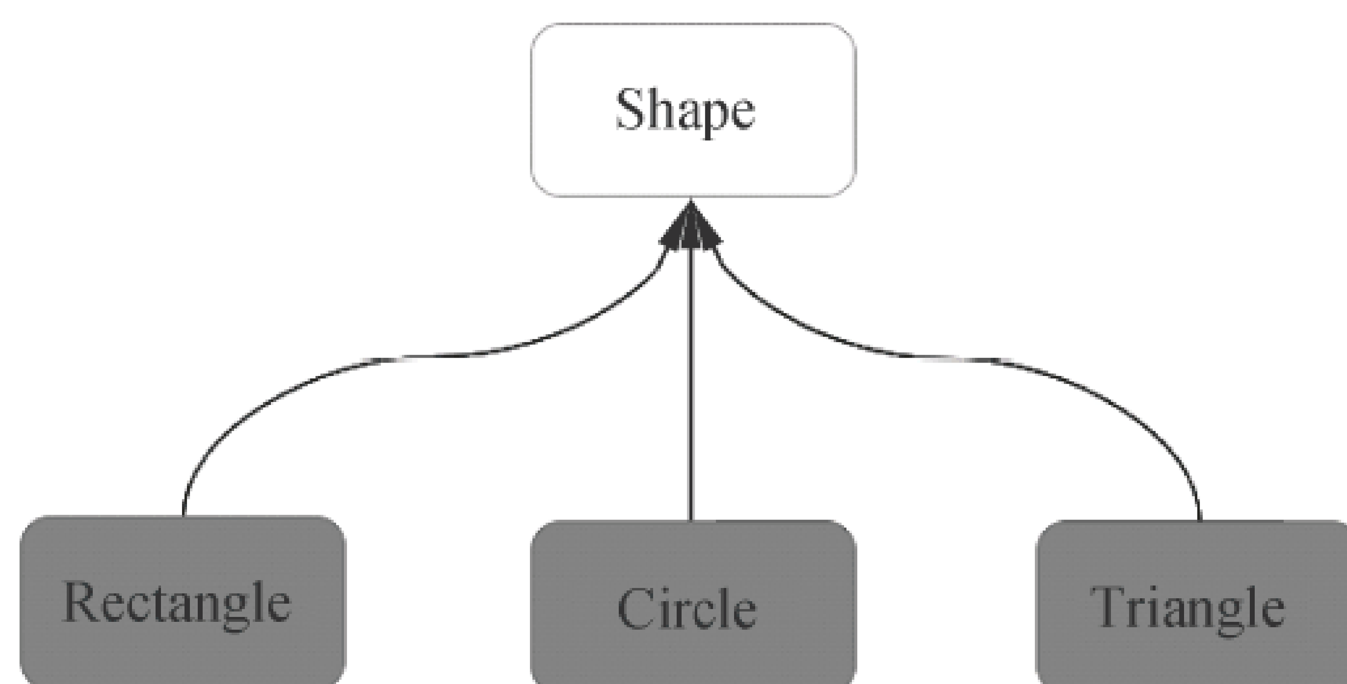


图 4-8　Shape 的抽象概念

对应的 Kotlin 代码如下：

```
package com.easy.kotlin

abstract class Shape          //声明抽象父类 Shape

class Rectangle : Shape()     //继承类的语法是使用冒号“:”，父类需要在这里使用构造
                                函数进行初始化

class Circle : Shape()        //Circle 继承 Shape 类

class Triangle : Shape()      //Triangle 继承 Shape 类
```

因为抽象的概念在问题领域中没有对应的具体概念，所以抽象类是不能够实例化的。下面的代码编译器会报错：

```
val s = Shape()               //编译不通过！不能实例化抽象类
```

我们只能实例化它的继承子类。代码示例如下：

```
val r = Rectangle()
println(r is Shape)           //true
```

现在我们有了抽象类，但是没有成员。通常，一个类的成员包括属性和函数。抽象类的成员也必须是抽象的，需要使用 abstract 关键字修饰。下面我们声明一个抽象类 Shape，并带有 width、heigth、radius 属性和 area()函数，代码如下：

```
abstract class Shape {
   abstract var width: Double
   abstract var heigth: Double
   abstract var radius: Double
   abstract fun area(): Double
}
```

这个时候，继承抽象类 Shape 的方法如下：

```
class Rectangle(override var width: Double, override var heigth: Double,
override var radius: Double) : Shape() {   //声明类的同时也声明了构造函数
    override fun area(): Double {
        return heigth * width
    }
}

class Circle(override var width: Double, override var heigth: Double,
override var radius: Double) : Shape() {
    override fun area(): Double {
        return 3.14 * radius * radius
    }
}
```

其中，override 是覆盖写父类属性和函数的关键字。在代码中可以这样调用具体实现的类函数：

```
fun main(args: Array<String>) {
    val r = Rectangle(3.0, 4.0, 0.0)
    println(r.area())                      //输出 12.0
    val c = Circle(0.0, 0.0, 4.0)
    println(c.area())                      //输出 50.24
}
```

抽象类中可以有带实现的函数，例如在抽象类 Shape 中添加一个函数 onClick()：

```
abstract class Shape {
    ...
    fun onClick() {                        //默认是 final 的，不可被覆盖重写
        println("I am Clicked!")
    }
}
```

那么在所有的子类中都可以直接调用这个 onClick()函数：

```
val r = Rectangle(3.0, 4.0, 0.0)           //声明 Rectangle 对象
r.onClick()                                //输出：I am Clicked!
val c = Circle(0.0, 0.0, 4.0)              //声明 Circle 对象
c.onClick()                                //输出：I am Clicked!
```

父类 Shape 中的 onClick()函数默认是 final 的，不可被覆盖重写。如果想要开放给子类重新实现这个函数，可以在前面加上 open 关键字：

```
abstract class Shape {
    ...
    open fun onClick() {
        println("I am Clicked!")
    }
}
```

在子类中这样覆盖重写：

```
class Rectangle(override var width: Double, override var heigth: Double,
override var radius: Double) : Shape() {//继承父类 Shape 的同时声明了构造函数
    override fun area(): Double {
        return heigth * width
    }
```

```
    override fun onClick(){
        println("${this::class.simpleName} is Clicked!")
    }
}

fun main(args: Array<String>) {
    val r = Rectangle(3.0, 4.0, 0.0)
    println(r.area())
    r.onClick()
}
```

其中，this::class.simpleName 是 Kotlin 中反射的 API，在 Gradle 工程的 build.gradle 中需要添加依赖 compile：

```
compile"org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"
```

关于反射的相关内容，将在第 12 章中详细介绍。

上面的代码运行后输出如下：

```
12.0
Rectangle is Clicked!
```

当子类继承了某个类之后，便可以使用父类中的成员变量，但并不是完全继承父类的所有成员变量。具体的原则如下：

- 能够继承父类的 public 和 protected 成员变量；
- 不能继承父类的 private 成员变量；
- 对于父类的包访问权限成员变量，如果子类和父类在同一个包下，则子类能够继承；否则，子类不能继承；
- 对于子类可以继承的父类成员变量，如果在子类中出现了同名称的成员变量，则会发生隐藏现象，即子类的成员变量会屏蔽掉父类的同名成员变量。如果要在子类中访问父类中的同名成员变量，需要使用 super 关键字进行引用。

4.3.2 接口

接口是一种比抽象类更加抽象的“类”。接口本身代表的是一种“类型”的概念。但在语法层面，接口本身不是类，不能实例化接口，只能实例化它的实现类。

接口是用来建立类与类之间的协议。实现接口的实现类必须要实现该接口的所有方法。在 Java 8 和 Kotlin 中，接口可以实现一些通用的方法。

接口是抽象类的延伸，Kotlin 与 Java 一样，不支持同时继承多个父类，也就是说继承只能存在一个父类（单继承）。但是接口不同，一个类可以同时实现多个接口（多组合），无论这些接口之间有没有关系。这样可以实现多重继承。

和 Java 类似，Kotlin 使用 interface 作为接口的关键词：

```
interface ProjectService    //Java 中使用 interface 声明接口
```

Kotlin 的接口与 Java 8 的接口类似。与抽象类相比，接口都可以包含抽象的方法及方法的实现：

```
interface ProjectService {
```

```
    val name: String
    val owner: String
    fun save(project: Project)
    fun print() {
        println("I am project")
    }
}
```

接口是没有构造函数的。我们使用冒号“:”语法来实现一个接口，如果有多个接口，用“，”逗号隔开：

```
class ProjectServiceImpl : ProjectService  //与继承抽象类语法一样，使用冒号
class ProjectMilestoneServiceImpl : ProjectService, MilestoneService
                                           //实现多个接口使用逗号“，”隔开
```

在重写 print()函数时，因为我们实现的 ProjectService、MilestoneService 都有一个 print()函数，当直接使用 super.print()函数时，编译器无法知道我们想要调用的是哪个 print 函数，我们把这种现象叫做覆盖冲突，如图 4-9 所示。

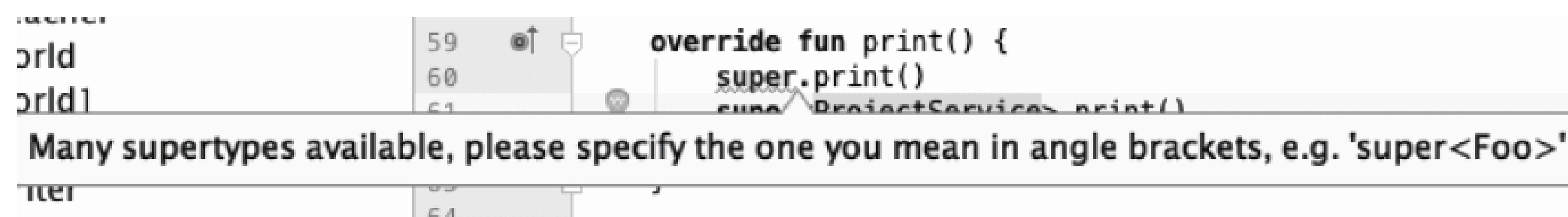


图 4-9　覆盖冲突

这个时候，我们可以使用下面的语法来调用：

```
super<ProjectService>.print()   //使用 super<ProjectService> 指定调用的是
                                  ProjectService 接口中的 print() 函数
super<MilestoneService>.print()//使用 super<MilestoneService> 来指定
                                  调用的是MilestoneService 接口中的 print()
                                  函数
```

4.4　object 对象

单例模式是一种常用的软件设计模式。例如，Spring 中的 Bean 默认就是单例。通过单例模式可以保证系统中一个类只有一个实例。即一个类只有一个对象实例。

Kotlin 中没有静态属性和方法，但是可以使用关键字 object 声明一个 object 单例对象：

```
package com.easy.kotlin

object User {                          //声明对象类型 User
    val username: String = "admin"
    val password: String = "admin"
    fun hello() {
        println("Hello, object !")
    }
}

fun main(args: Array<String>) {
    println(User.username)             //与 Java 静态类的调用形式一样
```

```
    println(User.password)
    User.hello()                    //与 Java 静态方法的调用方式一样
}
```

Kotlin 中还提供了伴生对象，用 companion object 关键字声明：

```
class DataProcessor {
    companion object DataProcessor { //使用 companion object 声明 DataProcessor
                                      的伴生对象
        fun process() {
            println("I am processing data ...")
        }
    }
}

fun main(args: Array<String>) {
    DataProcessor.process()         //I am processing data ...
}
```

一个类只能有一个伴生对象。

4.5 数 据 类

顾名思义，数据类就是只存储数据，不包含操作行为的类。Kotlin 中的数据类可以为我们节省大量的样板代码（Java 中强制我们要去写一堆 getter、setter 代码，而实际上这些方法都是“不言自明”的），这样最终的代码更易于理解，便于维护。

4.5.1 创建数据类

使用关键字为 data class 创建一个只包含数据的类：

```
data class LoginUser(val username: String, val password: String)
```

在 IDEA 中提供了方便的 Kotlin 工具箱，我们可以把上面的代码反编译成等价的 Java 代码。步骤如下：

（1）选择菜单栏中的 Tools|Kotlin|Show Kotlin Bytecode 命令，如图 4-10 所示。

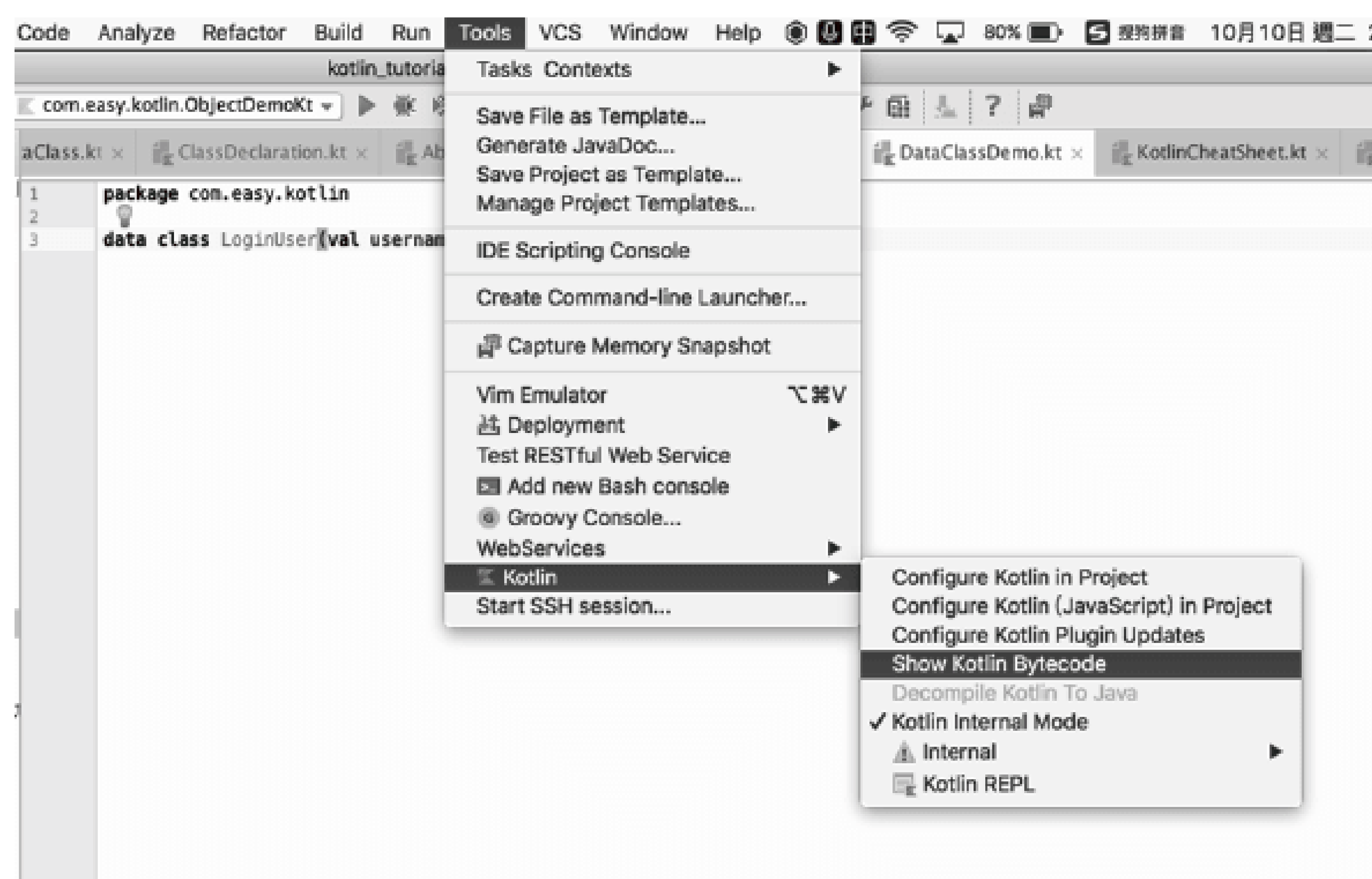


图 4-10　选择菜单栏中的 Show Kotlin Bytecode 命令

（2）在弹出的对话框中单击 Decompile 按钮，如图 4-11 所示。

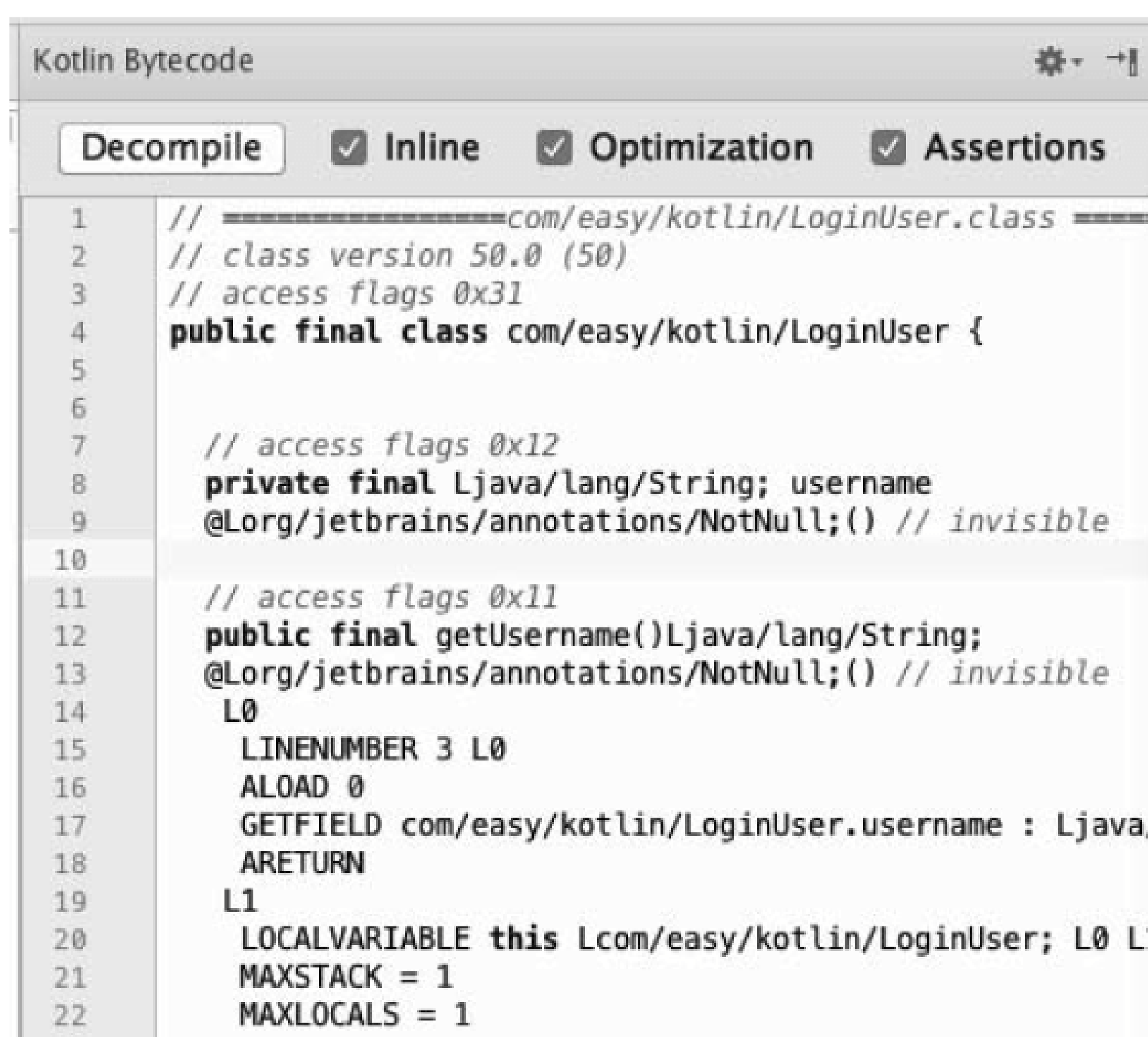


图 4-11　单击反编译按钮

（3）反编译之后的 Java 代码，如图 4-12 所示。

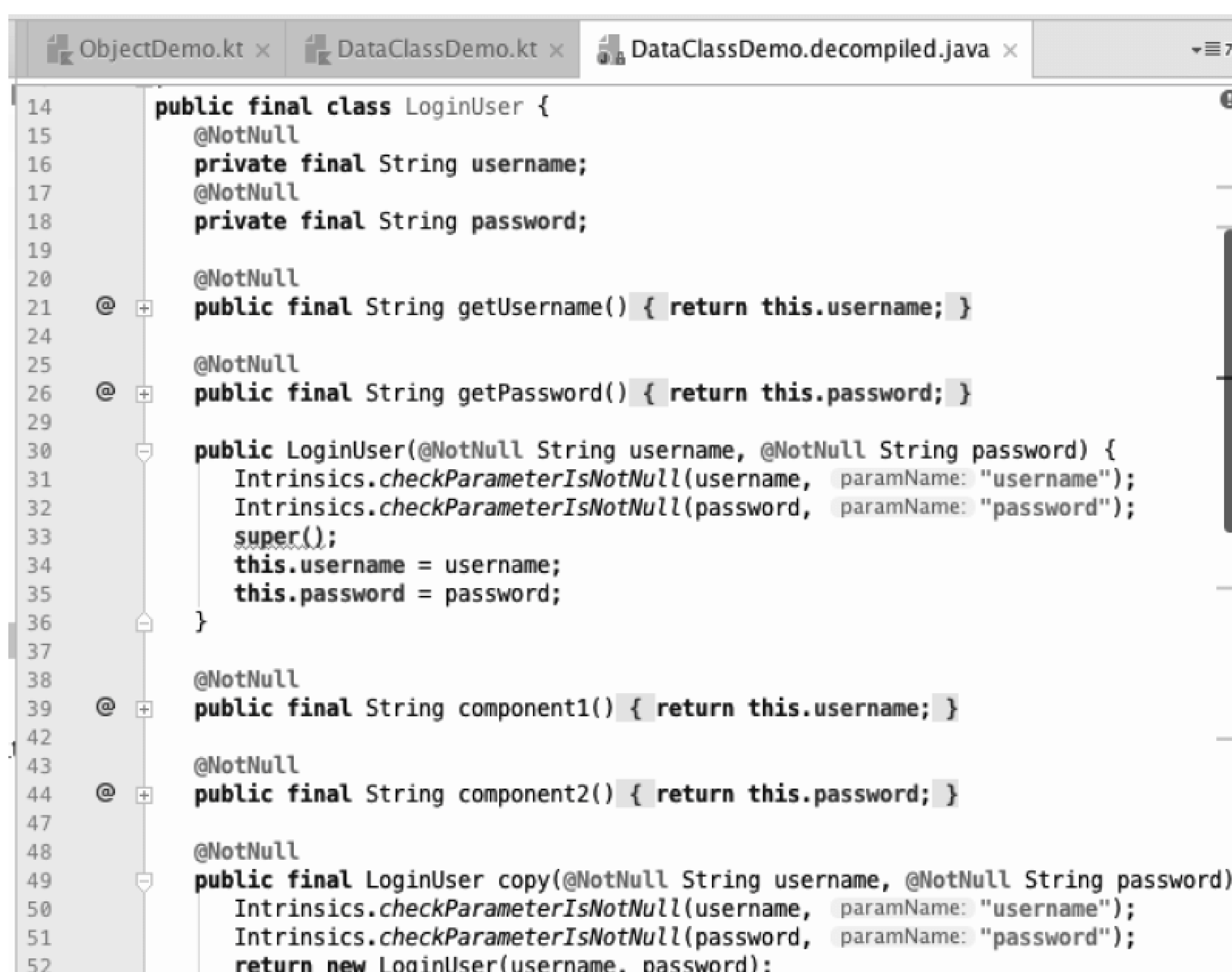


图 4-12　反编译之后的 Java 代码

上面这段代码反编译之后，完整的 Java 代码如下：

```
public final class LoginUser {
   @NotNull
   private final String username;
   @NotNull
   private final String password;

   @NotNull
```

```
public final String getUsername() {
   return this.username;
}

@NotNull
public final String getPassword() {
   return this.password;
}

public LoginUser(@NotNull String username, @NotNull String password) {
                                              //构造函数
   Intrinsics.checkParameterIsNotNull(username, "username");
   Intrinsics.checkParameterIsNotNull(password, "password");
   super();
   this.username = username;
   this.password = password;
}

@NotNull
public final String component1() {   //component1()函数，返回第 1 个成员值
                                        username
    return this.username;
}

@NotNull
public final String component2() {   //component2()函数，返回第 2 个成员值
                                        password
    return this.password;
}

@NotNull
public final LoginUser copy(@NotNull String username, @NotNull String
password) {
   Intrinsics.checkParameterIsNotNull(username, "username");
   Intrinsics.checkParameterIsNotNull(password, "password");
   return new LoginUser(username, password);
}

@NotNull
public static LoginUser copy$default(LoginUser var0, String var1, String
var2, int var3, Object var4) {
   if ((var3 & 1) != 0) {
      var1 = var0.username;
   }

   if ((var3 & 2) != 0) {
      var2 = var0.password;
   }

   return var0.copy(var1, var2);
}

public String toString() {
   return "LoginUser(username=" + this.username + ", password=" + this.
   password + ")";
}

public int hashCode() {
   return (this.username != null ? this.username.hashCode() : 0) * 31 +
   (this.password != null ? this.password.hashCode() : 0);
}
```

```
    public boolean equals(Object var1) {
        if (this != var1) {
            if (var1 instanceof LoginUser) {
                LoginUser var2 = (LoginUser)var1;
                if (Intrinsics.areEqual(this.username, var2.username) &&
                Intrinsics.areEqual(this.password, var2.password)) {
                    return true;
                }
            }
            return false;
        } else {
            return true;
        }
    }
}
```

4.5.2　数据类自动创建的函数

编译器会根据主构造函数中声明的属性，自动创建以下 3 个函数。

- equals()/hashCode() 函数 toString() 格式为"LoginUser(username="+this.username+", password="+this.password+")";
- component1()和 component2()函数返回对应下标的属性值，按声明顺序排列；
- copy()函数：根据旧对象属性重新 newLoginUser(username,password)一个对象出来。

如果这些函数在类中已经被明确定义了，或者从超类中继承而来，编译器就不再生成。

4.5.3　数据类的语法限制

数据类有如下限制：

- 主构造函数至少包含一个参数；
- 参数必须标识为 val 或者 var；
- 不能为 abstract、open、sealed 或者 inner；
- 不能继承其他类（但可以实现接口）。

另外，数据类可以在解构声明中使用：

```
package com.easy.kotlin

data class LoginUser(val username: String, val password: String)

fun main(args: Array<String>) {
    val loginUser = LoginUser("admin", "admin")
    val (username, password) = loginUser //解构声明 (username, password)
    println("username = ${username}, password = ${password}") //username =
    admin, password = admin
}
```

4.5.4　Pair 和 Triple

Kotlin 标准库提供了 Pair 和 Triple 数据类，分别表示二元组和三元组对象。它们的定

义分别如下：

```
public data class Pair<out A, out B>(
        public val first: A,
        public val second: B) : Serializable {
    public override fun toString(): String = "($first, $second)"
}
public infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
                                                    //中缀函数 to()

public data class Triple<out A, out B, out C>(
        public val first: A,
        public val second: B,
        public val third: C) : Serializable {
    public override fun toString(): String = "($first, $second, $third)"
}
```

我们可以使用 Pair 对象来初始化一个 Map，代码示例如下：

```
>>> val map = mapOf(1 to "A", 2 to "B", 3 to "C")
>>> map
{1=A, 2=B, 3=C}
```

4.6 注　　解

注解是将元数据附加到代码中。元数据信息由注解 kotlin.Metadata 定义。

```
@Retention(AnnotationRetention.RUNTIME)
@Target(AnnotationTarget.CLASS)
internal annotation class Metadata
```

这个@Metadata 信息存在于由 Kotlin 编译器生成的所有类文件中，并由编译器和反射读取。例如，使用 Kotlin 声明一个注解的代码如下：

```
annotation class Suspendable
```

Kotlin 中使用关键字 annotation **class** 来声明注解。

对应的 Java 代码如下：

```
@interface Suspendable
```

Kotlin 编译器会为注解生成对应的元数据信息：

```
@Retention(RetentionPolicy.RUNTIME)
@Metadata(
   mv = {1, 1, 7},
   bv = {1, 0, 2},
   k = 1,
   d1 = {"\u0000\n\n\u0002\u0018\u0002\n\u0002\u0010\u001b\n\u0000\b\
   u0086\u0002\u0018\u00002\u00020\u0001B\u0000¨\u0006\u0002"},
   d2 = {"Lcom/easy/kotlin/Suspendable;", "", "production sources for module
   kotlin_tutorials_main"}
)
public @interface Suspendable {
}
```

Kotlin 的注解完全兼容 Java 的注解。例如，在 Kotlin 中使用 Spring Data JPA 的代码示例如下：

```
interface ImageRepository : PagingAndSortingRepository<Image, Long> {

@Query("""
SELECT a from #{#entityName} a
where a.isDeleted=0
and a.isFavorite=1
and a.category like %:searchText%
order by a.gmtModified desc
""")          //(1) @Query 注解中 value 是一个 JPQL 字符串
fun  searchFavorite(@Param("searchText")  searchText:  String,  pageable:
Pageable):
Page<Image>     //(2) 声明一个查询接口，返回分页结果

@Throws(Exception::class)
@Modifying      //JPA 的保存/更新操作需要加上这个@Modifying 注解，表示这是一个修改操作
@Transactional  //JPA 的保存/更新操作需要加上这个@Transactional 注解，表示需要事务
@Query("update  #{#entityName}  a  set  a.isFavorite=1,a.gmtModified=now()
where a.id=?1")
fun addFavorite(id: Long)
}
```

代码说明如下：

- Kotlin 中使用 JPA 的@Query 注解，括号里面的参数是 JPQL 语句，我们使用 3 个双引号括起来，与 Python 中的语法一样。
- 使用@Param("searchText")注解来指定命名参数，在 JPQL 中使用 searchText 语法来使用这个参数。

可以看出，Kotlin 使用 Java 生态库中的注解，用起来与 Java 的注解基本一样。

下面再举一个 Kotlin 使用 SpringMVC 注解的代码例子。

```
@Controller          //(1) Kotlin 代码中直接使用@Controller 注解
class MeituController {
    @Autowired       //(2) Kotlin 代码中直接使用@ Autowired 注解
    lateinit var imageRepository: ImageRepository
                     //(3) 延迟初始化 imageRepository Bean
    //(4) Kotlin 代码中直接使用@RequestMapping 注解，需要注意这里关于 value，
      method 数组的语法与 Java 不同
    @RequestMapping(value = ["/", "meituView"], method = [RequestMethod.GET])
    fun meituView(model: Model, request: HttpServletRequest): ModelAndView {
        model["requestURI"] = request.requestURI //(5)
        return ModelAndView("meituView")
    }
}
```

代码说明如下：

- 第（1）处使用 SpringMVC 的@Controller 注解；
- 第（2）处使用 Spring 的@Autowired 注解装配 Bean；
- 第（3）处延迟初始化 Bean（等用到该 Bean 的时候才去创建对象）；
- 第（4）处使用 SpringMVC 的@RequestMapping 注解，其中，value 与 method 的值是数组，中括号[]的语法是 Kotlin1.2 中引入的特性。在这之前，我们需要使用

value=*arrayOf("/", "meituView")这样的语法。这个新语法特性背后的实现是 inline 函数，对应的实现代码如下：

```
public inline fun <reified @PureReifiable T> arrayOf(vararg elements: T):
Array<T>
```

其中，reified 是具体化类型关键字，@PureReifiable 注解用来指定对应的类型参数不能用于不安全操作，如强制转换或 is 检查。这意味着使用泛型类型作为参数是完全安全的。关于泛型将在第 8 章中介绍。

从上面的例子可以看出，在 Kotlin 中使用 Java 框架非常简便。

4.7 枚　　举

Kotlin 中使用 enum class 关键字来声明一个枚举类。例如：

```
enum class Direction {          //使用 enum class 声明一个 Direction 枚举类型
    NORTH, SOUTH, WEST, EAST    //每个枚举常量都是一个对象，用逗号分隔
}
```

相比于字符串常量，使用枚举能够实现类型安全。枚举类有两个内置的属性：

```
public final val name: String
public final val ordinal: Int
```

分别表示的是枚举对象的值与下标位置。例如上面的 Direction 枚举类，它的枚举对象的信息如下：

```
>>> val north = Direction.NORTH     //访问枚举中的 NORTH 对象
>>> north.name                      //name 属性
NORTH
>>> north.ordinal                   //ordinal 属性
0
>>> north is Direction              //north 的类型是 Direction
true
```

每一个枚举都是枚举类的实例，它们可以被初始化：

```
enum class Color(val rgb: Int) {    //声明一个带构造参数 rgb:Int 的枚举类
                                      Color(val rgb: Int)
    RED(0xFF0000),                  //rgb = 0xFF0000
    GREEN(0x00FF00),                //rgb = 0x00FF00
    BLUE(0x0000FF)                  //rgb = 0x0000FF
}
```

枚举 Color 的枚举对象信息如下：

```
>>> val c = Color.GREEN             //访问 Color 枚举类型中的 GREEN 元素
>>> c
GREEN
>>> c.rgb                           //访问 GREEN 枚举的 rgb 参数值
65280
>>> c.ordinal                       //访问 GREEN 枚举的 ordinal 属性
```

```
1
>>> c.name                          //GREEN 枚举的 name 属性
GREEN
```

4.8 内　部　类

本节我们介绍 Kotlin 的内部类，包括普通嵌套类、内部嵌套类和匿名内部类。

4.8.1　普通嵌套类

Kotlin 中，类可以嵌套。一个类可以嵌套在其他类中，而且可以嵌套多层。

```
class NestedClassesDemo {
    class Outer {
        private val zero: Int = 0
        val one: Int = 1

        class Nested {
            fun getTwo() = 2
            class Nested1 {
                val three = 3
                fun getFour() = 4
            }
        }
    }
}
```

测试代码如下：

```
val one = NestedClassesDemo.Outer().one
val two = NestedClassesDemo.Outer.Nested().getTwo()
val three = NestedClassesDemo.Outer.Nested.Nested1().three
val four = NestedClassesDemo.Outer.Nested.Nested1().getFour()
```

可以看出，代码中 NestedClassesDemo.Outer.Nested().getTwo()访问嵌套类的方式是直接使用类名.来访问，有多少层嵌套，就用多少层类名来访问。

普通嵌套类没有持有外部类的引用，所以是无法访问外部类变量的：

```
class NestedClassesDemo {
class Outer {
        private val zero: Int = 0
        val one: Int = 1

        class Nested {
            fun getTwo() = 2
            fun accessOuter() = {
                println(zero) //报错: cannot access outer class
                println(one)  //报错: cannot access outer class
            }
        }
}
}
```

4.8.2 嵌套内部类

如果一个类 Inner 想要访问外部类 Outer 中的成员，可以在这个类前面添加修饰符 inner。内部类会带有一个对外部类的对象引用。

```
package com.easy.kotlin

class NestedClassesDemo {
    class Outer {
        private val zero: Int = 0
        val one: Int = 1

        inner class Inner {      //使用 Inner 关键字声明内部类
            fun accessOuter() = {
                println(zero)      //works
                println(one)       //works
            }
        }
    }
}

fun main(args: Array<String>) {
    val innerClass = NestedClassesDemo.Outer().Inner().accessOuter()
}
```

可以看到，当访问 innerclassInner 的时候，我们使用的是 Outer().Inner()，这是持有了 Outer 的对象引用，与普通嵌套类直接使用类名访问的方式不同。

4.8.3 匿名内部类

匿名内部类就是没有名字的内部类。匿名内部类也可以访问外部类的变量。下面使用对象表达式创建一个匿名内部类实例：

```
class NestedClassesDemo {
    class AnonymousInnerClassDemo {
        var isRunning = false
        fun doRun() {
            Thread(object : Runnable {   //匿名内部类
                override fun run() {
                    isRunning = true
                    println("doRun : i am running, isRunning = $isRunning")
                }
            }).start()
        }
    }
}
```

如果对象是函数式 Java 接口，即具有单个抽象方法的 Java 接口的实例，例如上面例子中的 Runnable 接口：

```
@FunctionalInterface                    //Java 8 中引入的函数式接口注解
public interface Runnable {
    public abstract void run();         //函数式接口只有一个方法
}
```

我们可以使用 Lambda 表达式实现 Runnable 接口。下面的几种写法都是可以的：

```
fun doStop() {
    var isRunning = true
    Thread({                    //直接使用 Lambda 表达式
        isRunning = false
        println("doStop: i am not running, isRunning = $isRunning")
    }).start()
}

fun doWait() {
    var isRunning = true
    val wait = Runnable { //使用匿名内部类的方式，使用 Lambda 表达式实现 run 接口
        isRunning = false
        println("doWait: i am waiting, isRunning = $isRunning")
    }
    Thread(wait).start()
}

fun doNotify() {
    var isRunning = true
    val wait = {              //直接声明一个 Lambda 函数
        isRunning = false
        println("doNotify: i notify, isRunning = $isRunning")
    }
    Thread(wait).start()
}
```

更多关于 Lambda 表达式及函数式编程的相关内容，将在第 5 章中介绍。

4.9　本 章 小 结

本章我们介绍了 Kotlin 面向对象编程的特性：类与构造函数、抽象类与接口、继承与组合等知识，同时介绍了 Kotlin 中的注解类、枚举类、数据类、嵌套类、内部类、匿名内部类、单例 object 对象等特性类。

总的来说，在面向对象编程范式的支持上，Kotlin 相比于 Java，增加了不少有趣的功能与特性支持，这使得写起代码来更加方便、快捷了。

我们知道，在 Java 8 中，引进了对函数式编程的支持：Lambda 表达式、Function 接口、streamAPI 等，而在 Kotlin 中，对函数式编程的支持更加全面、丰富，代码写起来也更加简单、“优雅”。第 5 章中将一起学习 Kotlin 的函数式编程。

本章代码示例工程：https://github.com/EasyKotlin/kotlin_tutorials。

第 5 章　函数与函数式编程

凡此变数中含彼变数者，则此为彼之函数。（李善兰《代数学》）

函数式编程语言最重要的基础是 λ 演算（lambdacalculus），而且 λ 演算的函数可以传入函数参数，也可以返回一个函数。函数式编程（简称 FP）是一种编程范式（programming paradigm）。

函数式编程与命令式编程最大的不同是：函数式编程的焦点在于数据的映射，命令式编程（imperative programming）的焦点是解决问题的步骤。函数式编程不仅仅指的是 Lisp、Haskell、Scala 等类的语言，更重要的是一种编程思维，解决问题的思考方式，也称面向函数编程。

函数式编程的本质是函数的组合。例如，想要过滤出一个 List 中的奇数，用 Kotlin 代码可以这样写：

```
package com.easy.kotlin

fun main(args: Array<String>) {
    val list = listOf(1, 2, 3, 4, 5, 6, 7)
    println(list.filter { it % 2 == 1 }) //过滤函数，参数是一个 Lambda 表达式
}
```

这个映射的过程可以使用图 5-1 来形象地说明。

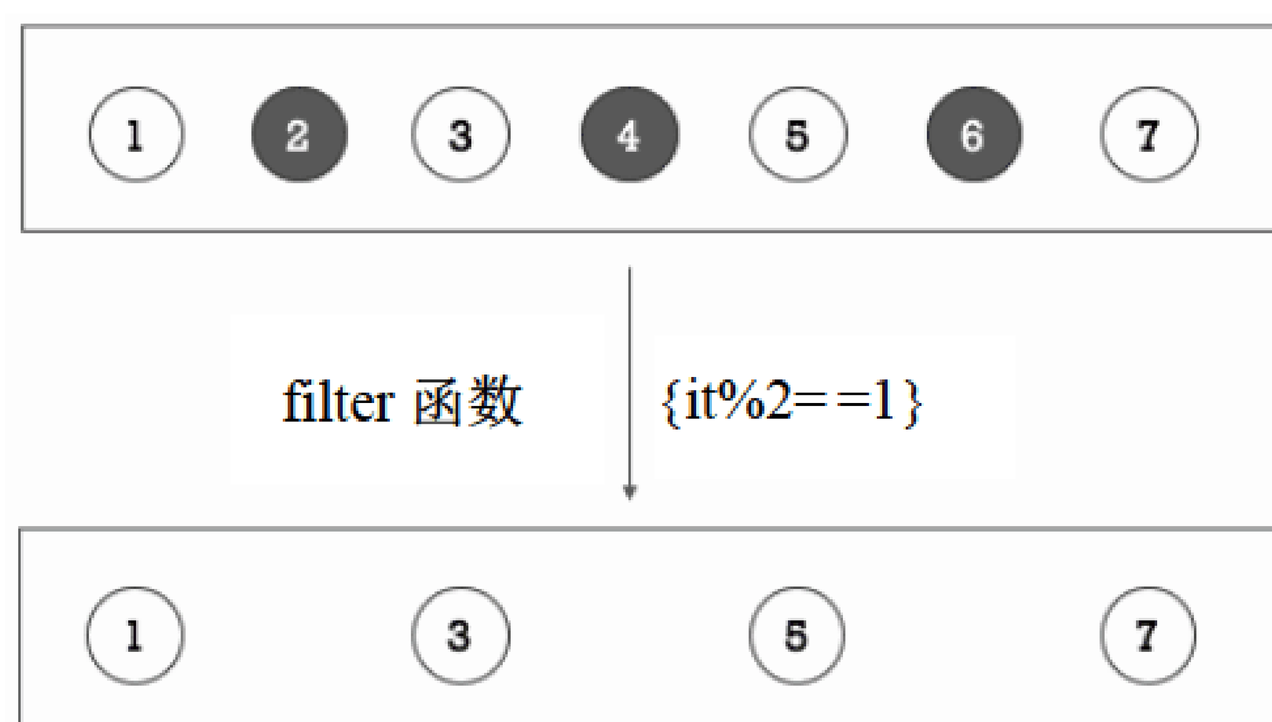


图 5-1　filter 函数的映射过程

而同样的逻辑使用命令式的思维方式来写的话，代码如下：

```
package com.easy.kotlin;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import static java.lang.System.out;
```

```
public class FilterOddsDemo {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(new Integer[] {1, 2, 3, 4, 5, 6, 7});
                                        //Java 中初始化 List
        out.println(filterOdds(list));  //输出：[1, 3, 5, 7]
    }

    public static List<Integer> filterOdds(List<Integer> list) {
                                            //使用命令式编程思维实现过滤函数
        List<Integer> result = new ArrayList();
        for (Integer i : list) {
            if (isOdd(i)) {
                result.add(i);
            }
        }
        return result;
    }

    private static boolean isOdd(Integer i) {  //判断是否是奇数的函数
        return i % 2 != 0;
    }
}
```

可以看出，函数式编程是简单、自然、直观易懂且美丽、“优雅”的编程风格。函数式编程语言中通常都会提供常用的 map、reduce、filter 等基本函数，这些函数是对 List、Map 集合等基本数据结构的常用操作的高层次封装，就像一个更加智能、好用的工具箱。

5.1　函数式编程简介

函数式编程是关于不变性和函数组合的编程范式。函数式编程有如下特征。

- 一等函数支持（first-class function）：函数也是一种数据类型，可以作为参数传入另一个函数中，同时函数也可以返回一个函数。
- 纯函数（pure function）和不变性（immutable）：纯函数指的是没有副作用的函数（函数不去改变外部的数据状态）。例如，一个编译器就是一个广义上的纯函数。在函数式编程中，倾向于使用纯函数编程。正因为纯函数不会去修改数据，同时又使用不可变的数据，所以程序不会去修改一个已经存在的数据结构，而是根据一定的映射逻辑创建一份新的数据。函数式编程是转换数据而非修改原始数据。
- 函数的组合（compose function）：在面向对象编程中是通过对象之间发送消息来构建程序逻辑的；而在函数式编程中是通过不同函数的组合来构建程序逻辑的。

5.2　声 明 函 数

Kotlin 中使用 fun 关键字来声明函数，其语法实例如图 5-2 所示。

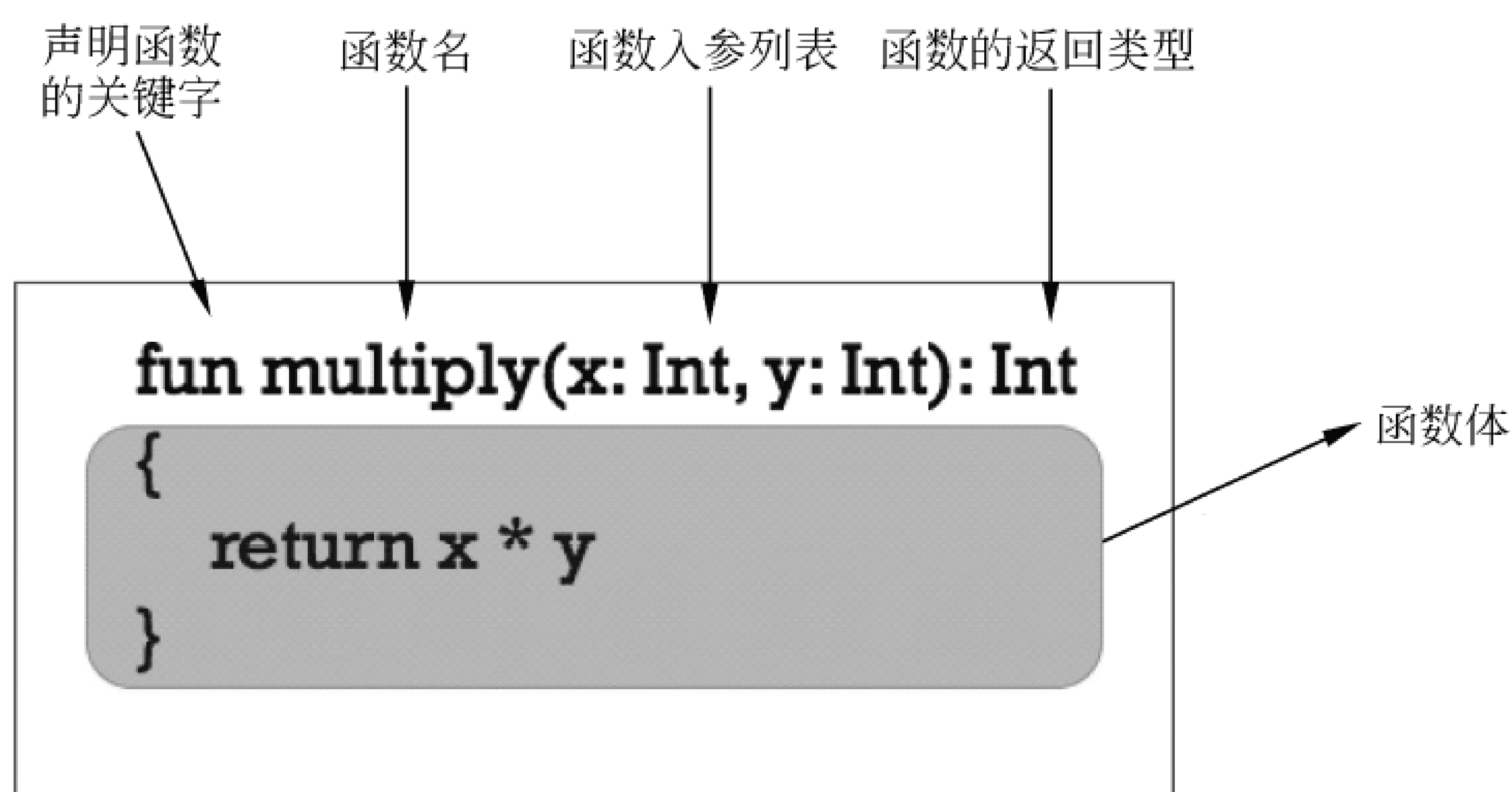


图 5-2　声明函数语法

为了更加直观地表现函数也可以当作变量来使用，声明一个函数类型的变量 sum 如下：

```
>>> val sum = fun(x:Int, y:Int):Int { return x + y }
                                            //sum 的类型是一个函数类型的变量
>>> sum
(kotlin.Int, kotlin.Int) -> kotlin.Int      //sum 是输入参数是 2 个 Int、输出类
                                              型是 Int 的函数
```

可以看到这个函数变量 sum 的类型是：

```
(kotlin.Int, kotlin.Int) -> kotlin.Int
```

这个带箭头"->"的表达式就是一个函数类型，表示一个输入两个 Int 类型值、输出一个 Int 类型值的函数。可以直接使用这个函数字面值 sum：

```
>>> sum(1,1) //直接使用 sum 这个函数字面值来调用函数
2
```

从上面这个典型的例子中可以看出，Kotlin 也是一种面向表达式的语言。既然 sum 是一个代表函数类型的变量，稍后我们将看到一个函数可以当作参数传入另一个函数中（高阶函数）。

当然，我们仍然可以像 C、C++、Java 语言一样，直接带上函数名来声明一个函数：

```
fun multiply(x: Int, y: Int): Int { //fun 关键字加上函数名 multiply 声明函数
    return x * y
}

multiply(2, 2) //4
```

5.3　Lambda 表达式

在本章开头部分讲到了下面这段代码：

```
val list = listOf(1, 2, 3, 4, 5, 6, 7)
list.filter { it % 2 == 1 }
```

这里的 filter()函数的入参{it%2==1}就是一段 Lambda 表达式。实际上，因为 filter()函数只有一个参数，所有括号被省略了。所以，filter()函数调用的完整写法是：

```
list.filter ({ it % 2 == 1 })
```

其中的 filter()函数声明如下：

```
public inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T>
                                                          //filter()函数签名
```

其实，filter()函数的入参是一个函数 predicate: (T) -> Boolean。实际上

```
{ it % 2 == 1 }                 //简写的 Lambda 表达式
```

是一种简写的语法，完整的 Lambda 表达式是这样写的：

```
{  it -> it % 2 == 1 }          //实际的 Lambda 表达式
```

如果拆开来写，就更加容易理解：

```
>>> val isOdd = { it: Int -> it % 2 == 1 }
            //直接使用 Lambda 表达式声明一个函数，这个函数判断输入的 Int 是不是奇数
>>> isOdd
(kotlin.Int) -> kotlin.Boolean      //isOdd 函数的类型
>>> val list = listOf(1, 2, 3, 4, 5, 6, 7)
>>> list.filter(isOdd)              //直接传入 isOdd 函数
[1, 3, 5, 7]
```

5.4　高 阶 函 数

本节介绍 Kotlin 中的高阶函数。

其实在上面的代码示例 list.filter(isOdd)中，已经看到了高阶函数。现在再添加一层映射逻辑。我们有一个字符串列表：

```
val strList = listOf("a", "ab", "abc", "abcd", "abcde", "abcdef", "abcdefg")
```

然后我们想要过滤出字符串元素的长度是奇数的列表。我们把这个问题的解决逻辑拆成两个函数来组合实现：

```
val f = fun (x: Int) = x % 2 == 1       //判断输入的 Int 是否奇数
val g = fun (s: String) = s.length      //返回输入的字符串参数的长度
```

我们再使用函数 h 来封装“字符串元素的长度是奇数”这个逻辑，实现代码如下：

```
val h = fun(g:  (String) -> Int, f: (Int) -> Boolean):  (String) -> Boolean {
    return { f(g(it)) }
}
```

但是这个 h 函数的声明有些长了，尤其是 3 个函数类型声明的箭头表达式，显得不够简洁。不过不用担心，Kotlin 中有简单好用的 Kotlin 类型别名，我们使用 G、F、H 来声明 3 个函数类型：

```
typealias G = (String) -> Int
typealias F = (Int) -> Boolean
typealias H = (String) -> Boolean
```

那么，我们的 h 函数就可以写成下面这样了：

```
val h = fun(g: G, f: F): H {
    return { f(g(it)) } //需要注意的是，这里的 {} 是不能省略的
}
```

这个 h 函数的映射关系可用图 5-3 来说明。

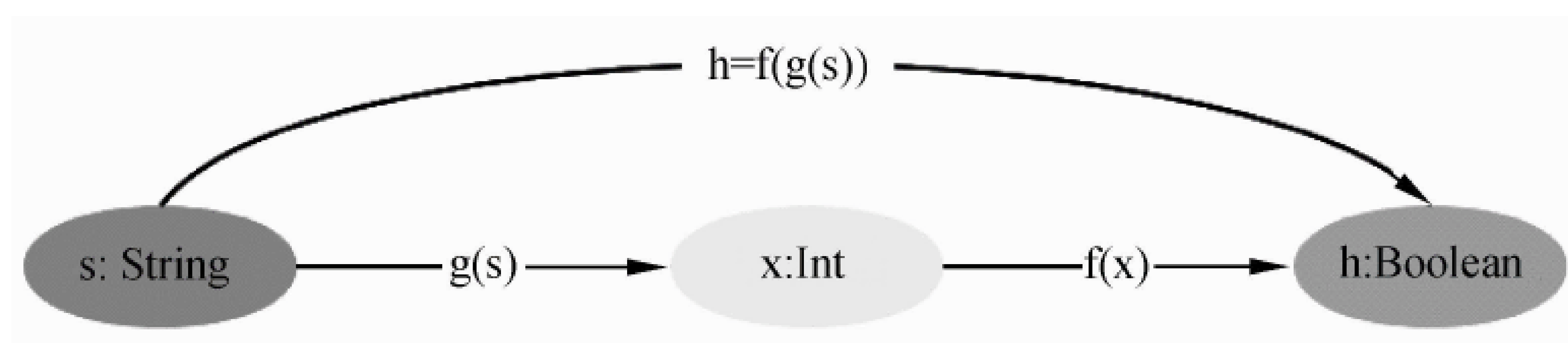


图 5-3　复合函数 h 的映射关系

在函数体的代码 return{ f(g(it))}中，{}代表这是一个 Lambda 表达式，返回的是一个(String)->Boolean 函数类型。如果没有{}，那么返回值就是一个布尔类型 Boolean 了。

通过上面的代码例子可以看到，在 Kotlin 中，我们可以简单地实现高阶函数。现在逻辑已经实现完成，下面我们在 main()函数中运行测试一下效果。

```
fun main(args: Array<String>) {
    val strList = listOf("a", "ab", "abc", "abcd", "abcde", "abcdef", "abcdefg")
    println(strList.filter(h(g, f)))    //输出: [a, abc, abcde, abcdefg]
}
```

当你看到 h(g,f)这样的复合函数的代码时一定很开心，感到很自然，这与数学公式很贴近，简单易懂。

5.5　Kotlin 中的特殊函数

本节我们介绍 Kotlin 中的 run()、apply()、let()、also()和 with()这 5 个特殊的函数。本节代码示例中用到的测试函数 myfun()代码如下：

```
fun myfun(): String {
    println("执行了 myfun 函数")
    return "这是 myfun 的返回值"
}
```

5.5.1　run()函数

run()函数的定义如下：

```
public inline fun <R> run(block: () -> R): R {
    contract {
```

```
        callsInPlace(block, InvocationKind.EXACTLY ONCE)
    }
    return block()
}
```

我们重点看最后一行代码 block()，其实就是调用传入的 block 参数，一般情况下是一个 Lambda 代码块。测试代码示例如下：

```
fun testRunFun() {
    myfun()                     //直接在代码行调用函数
    run({ myfun() })            //使用 run()函数调用 myfun()函数
    run { myfun() }             //run()函数的括号“()”可以省略
    run { println("A") }        //等价于 println("A")
}

fun main(args: Array<String>) {
    testRunFun()
}
```

运行上面的代码，输出如下：

```
执行了 myfun 函数
执行了 myfun 函数
执行了 myfun 函数
A
```

5.5.2　apply()函数

apply()函数的定义如下：

```
public inline fun <T> T.apply(block: T.() -> Unit): T {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY ONCE)
    }
    block()
    return this
}
```

同样，我们重点看最后两行代码，先是调用了 block()函数，然后返回当前的调用者对象 this。意思是执行完 block()代码块逻辑后，再次返回当前的调用者对象。测试代码示例如下：

```
fun testApply() {
    //普通写法
    val list = mutableListOf<String>()
    list.add("A")
    list.add("B")
    list.add("C")
    println("普通写法 list = $list")          //普通写法 list = [A, B, C]
    println(list)

    //使用 apply()函数的写法
    val a = ArrayList<String>().apply {      //调用 apply()函数
```

```
        add("A")
        add("B")
        add("C")
        println("使用 apply 函数写法 this = $this")
    }
    println(a)
    //等价于
    a.let { println(it) }
}

fun main(args: Array<String>) {
    testApply()
}
```

运行上面的代码，输出如下：

```
普通写法 list = [A, B, C]
[A, B, C]
使用 apply 函数写法 this = [A, B, C]
[A, B, C]
[A, B, C]
```

5.5.3 let()函数

let()函数的定义如下：

```
public inline fun <T, R> T.let(block: (T) -> R): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return block(this)
}
```

同样，我们还是重点看最后一行代码 block(this)，意思是把当前调用对象作为参数传入 block()代码块中。测试代码示例如下：

```
fun testLetFun() {
    1.let { println(it) }            //输出 1，其中 it 就是调用者 1
    "ABC".let { println(it) }        //输出 ABC，其中 it 就是调用者 ABC
    //执行完函数 myfun()，返回值传给 let()函数
    myfun().let {
        print(it)
    }
}

fun main(args: Array<String>) {
    testLetFun()
}
```

运行上面的代码，输出如下：

```
1
ABC
执行了 myfun 函数
```

```
这是 myfun 的返回值
```

5.5.4　also()函数

also()函数的定义如下：

```
public inline fun <T> T.also(block: (T) -> Unit): T {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    block(this)
    return this
}
```

同样，我们还是看最后两句，首先是调用了 block(this)，类似 let()函数的逻辑，但是最后返回的值是 this，也就是当前的调用者。测试代码示例如下：

```
fun testAlsoFun() {
    val a = "ABC".also {
        println(it)  //输出：ABC
    }
    println(a)       //输出：ABC
    a.let {
        println(it)  //输出：ABC
    }
}

fun main(args: Array<String>) {
    testAlsoFun()
}
```

5.5.5　with()函数

with()函数的定义如下：

```
public inline fun <T, R> with(receiver: T, block: T.() -> R): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return receiver.block()
}
```

我们看到 with()函数传入了一个接收者对象 receiver，然后使用该对象 receiver 去调用传入的 Lambda 代码块 receiver.block()。测试代码如下：

```
fun testWithFun() {
    //普通写法
    val list = mutableListOf<String>()
    list.add("A")
    list.add("B")
    list.add("C")
    println("常规写法 list = $list")      //常规写法 list = [A, B, C]

    //使用 with()函数写法
    with(ArrayList<String>()) {
```

```
        add("A")
        add("B")
        add("C")
        println("使用 with 函数写法 this = $this")
                                        //使用 with()函数的写法 this = [A, B, C]
    }.let {
        println(it) //kotlin.Unit
    }
}

fun main(args: Array<String>) {
    testWithFun()
}
```

5.6 本 章 小 结

在 Kotlin 中，支持函数作为“一等公民”，它支持高阶函数、Lambda 表达式等。我们不仅可以把函数当作普通变量一样传递、返回，还可以把它分配给变量、放进数据结构或者进行一般性的操作。在 Kotlin 中进行函数式编程相当简单。

第 6 章　扩展函数与属性

在使用 Java 的时候，我们经常使用诸如 StringUtil、DateUtil 等工具类，代码写起来比较冗长。举个例子，获取一个字符串的第一个字符值、最后一个字符值。如果我们用 Java 代码来写，通常是要先声明一个 StringUtil 类，然后在里面写相应的工具方法，代码可以是下面这样：

```
package com.easy.kotlin;

import static java.lang.System.out;

public class StringUtil {                    //Java 工具类 StringUtil 的代码

    /**
     * 获取 str 的第一个字符值
     *
     * @param str
     * @return
     */
     public static String firstChar(String str) {
       if (str != null && str.length() > 0) {//非空判断
           return str.charAt(0) + "";        //获取第 1 个字符，转为 String 返回
       }
       return "";
    }
    /**
     * 获取 str 的最后一个字符值
     *
     * @param str
     * @return
     */
    public static String lastChar(String str) {
       if (str != null && str.length() > 0) {
           return str.charAt(str.length()-1)+"";//获取最后一个字符，转为 String 返回
       }
       return "";
}

    public static void main(String[] args) {
       String str = "abc";
       out.println(StringUtil.firstChar(str));     //返回 a
       out.println(StringUtil.lastChar(str));      //返回 c
    }
}
```

我们可以看到 StringUtil.firstChar(str)这样的调用方式不够简单直接。能不能直接这样调用呢？

```
"abc".firstChar()
"abc".lastChar()
```

非常遗憾的是，在 Java 中我们无法给 String 类添加一个自定义方法。因为 String 类是

JDK 中内置的基础类，而且为 final，不能修改。所以，Java 程序员通常使用这样一个变通的方法：开发一个 StringUtil 类，在里面封装所需要的 String 操作的方法，而不是修改或继承 String 类。

而在 Kotlin 里，情况就完全不一样了——我们完全可以自由扩展任何类的方法和属性。在不修改原类的情况下，Kotlin 能给一个类扩展新功能而无须继承该类。

本章我们将介绍 Kotlin 的扩展函数和属性。

6.1 扩 展 函 数

Kotlin 中提供了使用非常简单的扩展函数功能。我们可以为现有的类自由添加自定义的函数。

6.1.1 给 String 类扩展两个函数

例如，现在给 String 类扩展两个函数 firstChar()和 lastChar()，实现代码如下：

```
package com.easy.kotlin

fun String.firstChar(): String {//Kotlin 中给 String 扩展一个 firstChar 函数
    if (this.length == 0) {
        return ""
    }
    return this[0].toString()
}

fun String.lastChar(): String { //Kotlin 中给 String 扩展一个 lastChar 函数
    if (this.length == 0) {
        return ""
    }
    return this[this.length - 1].toString()
}
```

扩展函数的语法可以用图 6-1 来简单说明。

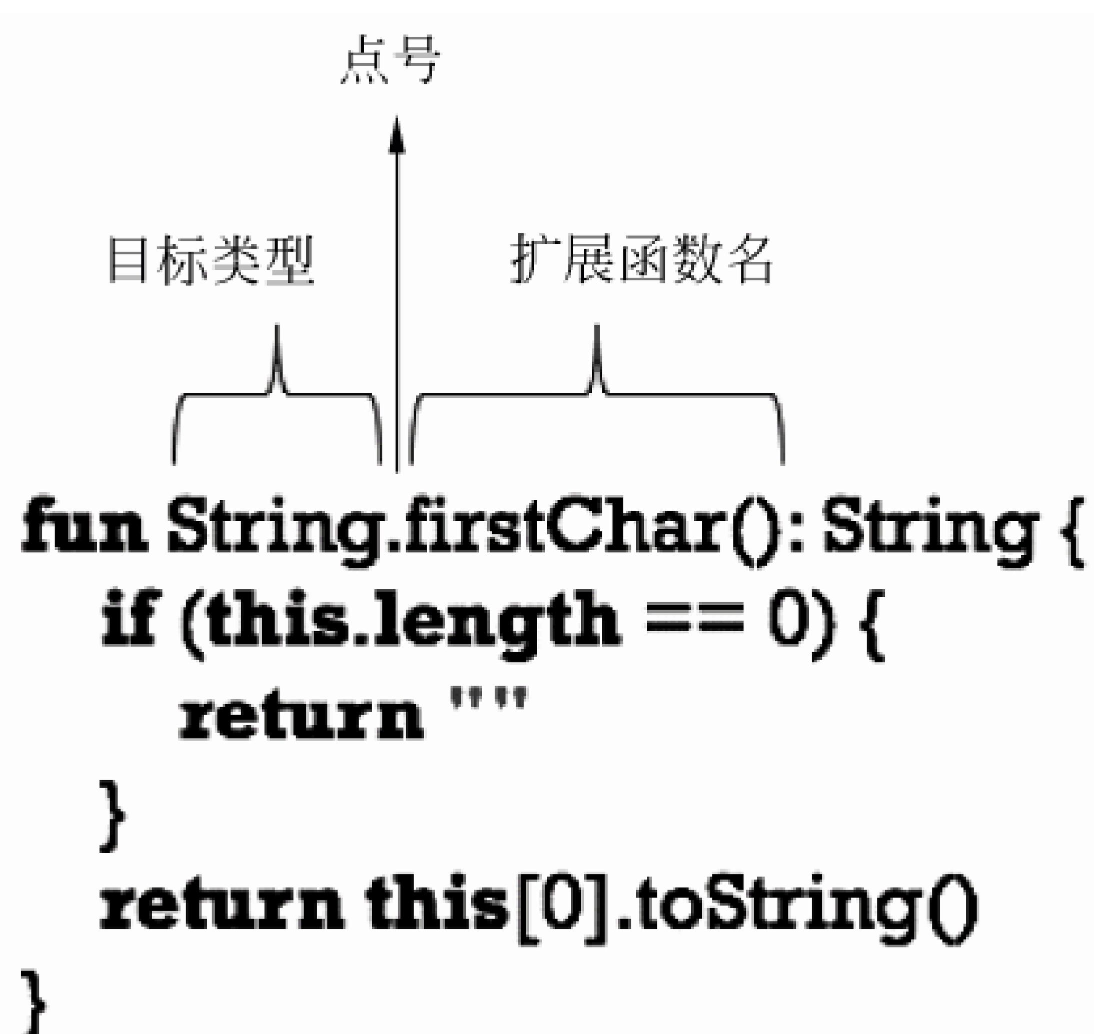


图 6-1　给 String 类型扩展一个 firstChar()函数

然后就可以在代码中直接调用了：

```
fun main(args: Array<String>) {
    println("abc".firstChar())       //返回 a
    println("abc".lastChar())        //返回 c
}
```

如果在其他 package 路径下面，则需要 import 导入扩展函数：

```
package com.easy.kotlin.tutorial   //与扩展函数不在同一个包路径下

import com.easy.kotlin.firstChar   //导入扩展函数 firstChar()
import com.easy.kotlin.lastChar    //导入扩展函数 lastChar()

fun main(args: Array<String>) {
    val str = "abc"
    str.firstChar() //这样的调用方式要比 StringUtil.firstChar(str) 简单许多
    str.lastChar()
}
```

6.1.2　给 List 类扩展一个过滤函数

在第 5 章中我们介绍过 List 的 filter()函数。那么这个 filter()函数是怎样实现的呢？ 如果我们自己给 List 类扩展一个过滤函数，应该怎样去做呢？下面我们就来解决这个问题。

为了让读者能更加深刻地体会到 Kotlin 扩展功能的简单、优雅性，我们先来看看在 Java 中是怎样实现的吧！首先，我们会去声明一个 ListUtil 类，里面实现一个 List filter(List list,Predicatep)方法，代码如下：

```
public class ListUtil<T> {
    /**
     * 根据谓词 p 过滤 list 中的元素
     *
     * @param list
     * @param p
     * @return
     */
    public List<T>filter(List<T>list,Predicate<T> p){
                                              //Java 中的 filter()方法的实现
        List<T> result = new ArrayList<>();
        for (T t : list) {
            if (p.predicate(t)) {             //如果满足判定条件
                result.add(t);                //添加该元素到 result()列表中
            }
        }
        return result;
    }
}
```

其中，Predicate 接口声明如下：

```
interface Predicate<T> {
    Boolean predicate(T t);                   //返回布尔值的谓词函数
}
```

然后，我们在代码中这样使用这个 filter()方法：

```
public static void main(String[] args) {
    List<Integer> list = Arrays.asList(new Integer[] {1, 2, 3, 4, 5, 6, 7});
    ListUtil<Integer> listUtil = new ListUtil();   //声明 ListUtil 对象
    List<Integer> result = listUtil.filter(list, (it) -> it % 2 == 1);
                                                   //Lambda 表达式
    out.println(result);                           //输出: [1, 3, 5, 7]
}
```

为了调用 filter()方法，我们还要声明一个 ListUtil 对象，这样显得比较麻烦。能不能直接像下面这样调用呢？

```
list.filter { it % 2 == 1 }
```

答案是肯定的，只不过必须是在 Kotlin 中，而不是在 Java 中。下面我们就来使用 Kotlin 中的扩展函数为 List 扩展一个 filter()函数，代码如下：

```
fun <T> List<T>.filter(predicate: (T) -> Boolean): MutableList<T> {
                                  //Koltin 中给 List 类扩展一个 filter 函数
    val result = ArrayList<T>()
    this.forEach {                //这里的 this 指向调用者对象
        if (predicate(it)) {      //如果满足谓词判断条件
            result.add(it)
        }
    }
    return result
}
```

这个函数的签名稍微有点复杂，我们用图 6-2 来形象化地简单说明。

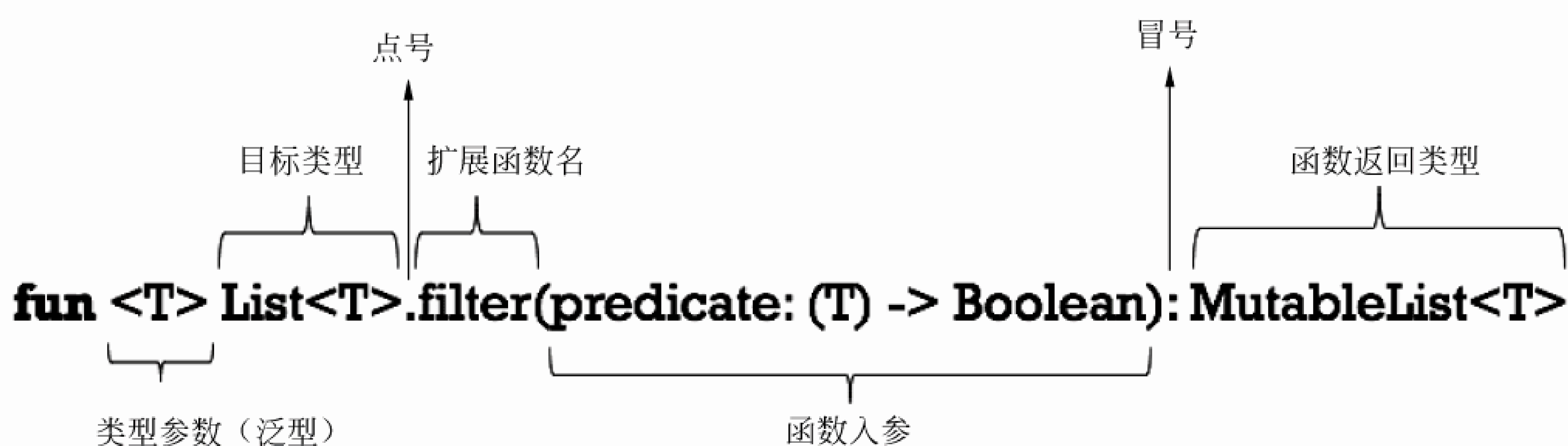


图 6-2　filter()函数的签名

然后我们在代码中只需要这样调用即可：

```
val list = mutableListOf(1, 2, 3, 4, 5, 6, 7)
val result = list.filter {
    it % 2 == 1       //这是一个 Lambda 函数
}
println(result)       //[1, 3, 5, 7]
```

Kotlin 的标准库 API 中使用了扩展的功能，通过扩展 Java 的 API，提供了大量实用且简单的函数，这部分内容将在第 9 章中具体介绍。

6.2 扩 展 属 性

除了扩展一个类的函数，还可以扩展类属性。例如，我们给 MutableList 扩展两个属性：firstElement 和 lastElement，实现代码如下：

```
var <T> MutableList<T>.firstElement: T
    get() {                        //扩展属性 firstElement 的 get()函数
        return this[0]             //返回第 1 个元素
    }
    set(value) {                   //扩展属性 firstElement 的 set()函数
        this[0] = value            //设置第 1 个元素为 value
    }
    var <T> MutableList<T>.lastElement: T
    get() {
        return this[this.size - 1] //返回最后一个元素
    }
    set(value) {
        this[this.size - 1] = value //设置最后一个元素的值为 value
    }
```

上面代码中扩展属性的语法说明如图 6-3 所示。

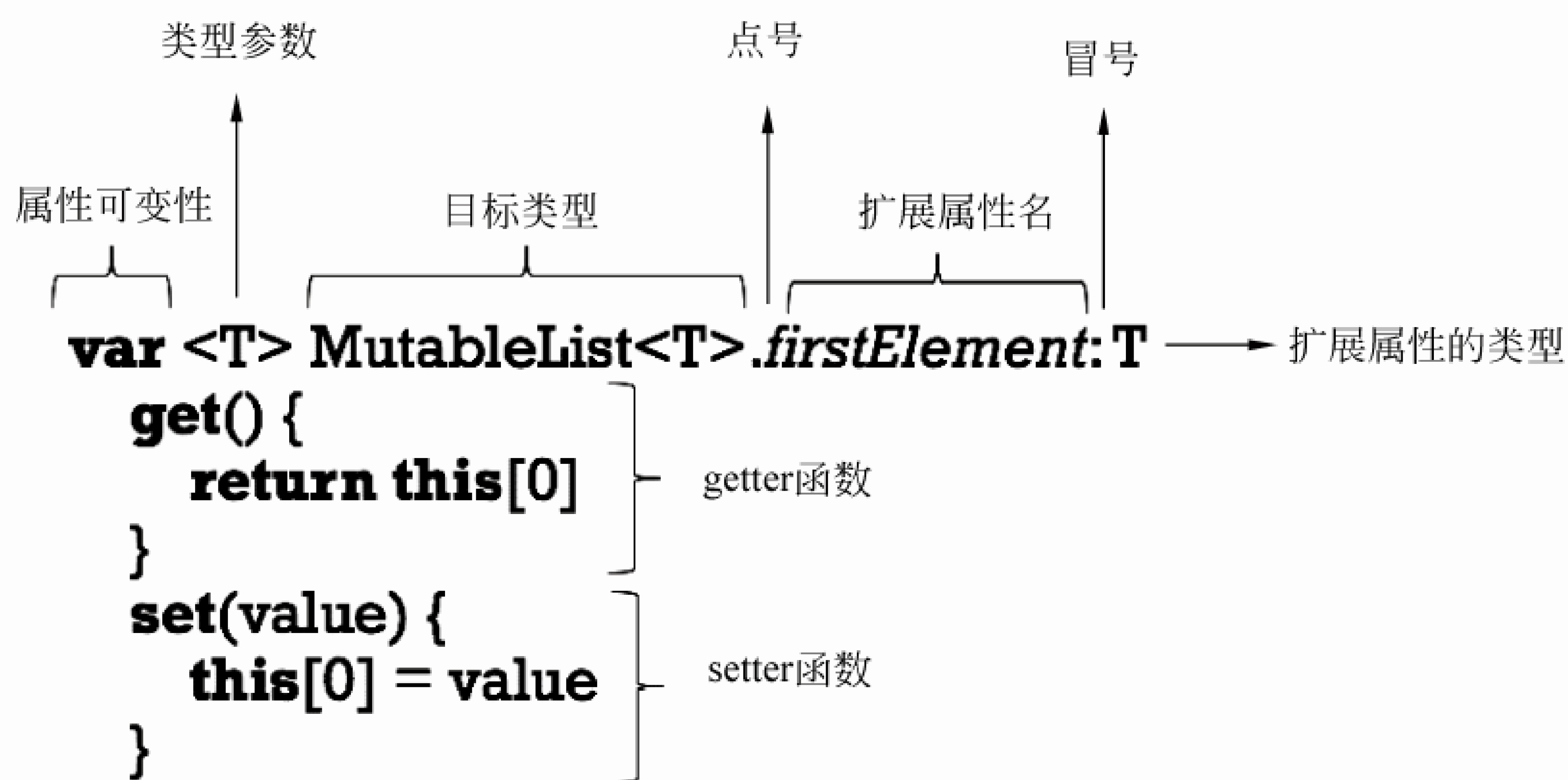


图 6-3　扩展属性的语法说明

然后就可以在代码中直接使用扩展的属性了：

```
val list = mutableListOf(1, 2, 3, 4, 5, 6, 7)

println("list = ${list}")   //list = [1, 2, 3, 4, 5, 6, 7]
println(list.firstElement)  //调用 getter 函数，值是 1
println(list.lastElement)   //7

list.firstElement = -1      //调用 setter 函数
list.lastElement = -7

println("list = ${list}")   //list = [-1, 2, 3, 4, 5, 6, -7]
println(list.firstElement)  //-1
println(list.lastElement)   //-7
```

扩展属性允许定义在类或者 Kotlin 文件中，不允许定义在函数中。

6.3 扩展的实现原理

扩展属性和扩展函数的本质是以静态导入的方式来实现的。其背后的实现原理可以通过 Kotlin 代码的 ByteCode 来理解。例如我们在 6.1.1 节中给 String 类型扩展的 firstChar() 函数：

```
fun String.firstChar(): String {
    if (this.length == 0) {
        return ""
    }
    return this[0].toString()
}
```

它对应的 JVM 码如下：

```
  //access flags 0x19
  public final static firstChar(Ljava/lang/String;)Ljava/lang/String;
  @Lorg/jetbrains/annotations/NotNull;()   //invisible
   @Lorg/jetbrains/annotations/NotNull;() //invisible, parameter 0
   L0
    ALOAD 0
    LDC "$receiver"
    INVOKESTATIC kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull
(Ljava/lang/Object;Ljava/lang/String;)V
   L1
    LINENUMBER 4 L1
    ALOAD 0
    INVOKEVIRTUAL java/lang/String.length ()I
    IFNE L2
   L3
    LINENUMBER 5 L3
    LDC ""
    ARETURN
   L2
    LINENUMBER 7 L2
    ALOAD 0
    ICONST 0
    INVOKEVIRTUAL java/lang/String.charAt (I)C
    INVOKESTATIC java/lang/String.valueOf (C)Ljava/lang/String;
    ARETURN
   L4
    LOCALVARIABLE $receiver Ljava/lang/String; L0 L4 0
    MAXSTACK = 2
    MAXLOCALS = 1
```

直接看上面的 JVM 指令可能不直观，反编译成 Java 代码后会更清楚：

```
public static final String firstChar(@NotNull String $receiver) {
   Intrinsics.checkParameterIsNotNull($receiver, "$receiver");
   return $receiver.length() == 0 ? "" : String.valueOf($receiver. charAt(0));
}
```

6.4 扩展中的 this 关键字

在前面的 List 扩展函数 filter()的实现中，用到了一个 this 关键字：

```
this.forEach {            //this 关键字
    if (predicate(it)) {
        result.add(it)
    }
}
```

这里的 this 指的是接收者对象（receiver object），也就是调用扩展函数时，在点号“.”之前指定的对象实例。为了表示当前函数的接收者（receiver），Kotlin 中使用 this 表达式：

- 在类的成员函数中，this 指向这个类的当前对象实例；
- 在扩展函数中，或带接收者的函数字面值（function literal）中，this 代表调用函数时，在点号左侧传递的接收者参数；
- 如果 this 没有限定符，那么它指向包含当前代码的最内层范围。如果想要指向其他范围内的 this，需要使用标签限定符。

编程技巧提示： 可以新建一个公共源文件，把自定义的扩展属性和扩展函数都放到包中，作为一个通用工具类来使用。

6.5 本章小结

扩展函数是 Kotin 中非常方便且实用的功能，使用扩展函数，可以使我们的代码写起来更加简单。同时也正是通过这个特性，使 Kotlin 在 Java API 的基础上扩展了丰富实用的函数，我们将在后面的章节中具体介绍。

第 7 章 集 合 类

在 Java 类库中有一套相当完整的容器集合类来持有对象。Kotlin 没有去重复造轮子（Scala 则是自己实现了一套集合类框架），而是在 Java 类库的基础上进行了改造和扩展，引入了不可变集合类，同时扩展了大量方便实用的功能，这些功能的 API 都在 kotlin.collections 包下面。

另外，在 Kotlin 的集合类中不仅仅能持有普通对象，而且能够持有函数类型的变量。例如下面是一个持有两个函数的集合类：

```
val funlist: List<(Int) -> Boolean>=//声明一个持有类型为函数 (Int) -> Boolean 的 List
            listOf({ it -> it % 2 == 0 },   //第 1 个函数为{ it -> it % 2 == 0 }
                   { it -> it % 2 == 1 })   //第 2 个函数为{ it -> it % 2 == 1 }
```

其中，(Int)->Boolean 是一个从 Int 映射到 Boolean 的函数。而这个时候，我们可以在代码里选择调用哪个函数：

```
val list = listOf(1, 2, 3, 4, 5, 6, 7)
list.filter(funlist[0]) //传入第 1 个函数 funlist[0],返回[2, 4, 6]
list.filter(funlist[1]) //传入第 2 个函数 funlist[1],返回[1, 3, 5, 7]
```

是不是感觉很有意思？这就是面向对象范式混合函数式编程的自由乐趣吧！

本章将介绍 Kotlin 标准库中的集合类，我们将了解到它是如何扩展 Java 集合库的，使代码写起来更加简单、容易。

7.1 集合类概述

集合类存放的都是对象的引用，而非对象本身，我们通常说的集合中的对象指的是集合中对象的引用（reference)。

Kotlin 的集合类分为：可变集合类（Mutable）与不可变集合类（Immutable）。

7.1.1 常用的 3 种集合类

集合类主要有 3 种：List（列表）、Set（集）和 Map（映射），如图 7-1 所示。

List 容器中的元素以线性方式存储，集合中可以存放重复对象。列表中的元素是有序

地排列。

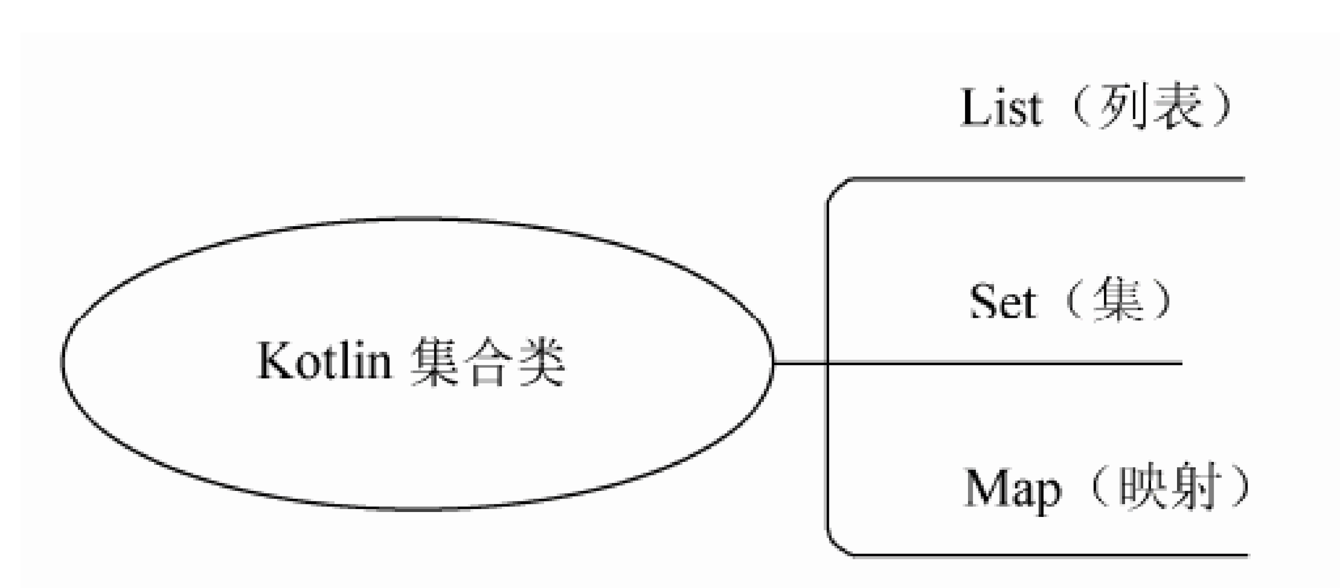


图 7-1　Kotlin 集合类

Set 集容器的元素无序、不重复。

Map 映射中持有的是“键值对”对象，每一个对象都包含一对键值 K-V 对象。Map 映射容器中存储的每个对象都有一个相关的关键字（Key）对象，关键字决定对象在映射中的存储位置。关键字是唯一的。其实关键字本身并不能决定对象的存储位置，它通过散列（hashing）产生一个被称做散列码（hash code）的整数值，这个散列码对应值（Value）的存储位置。

如果我们从数据结构的本质上来看，其实 List 中的下标就是 Key，只不过 Key 是有序的 Int 类型，所以说 List 也可以说是一种特殊的 Map 数据结构。而 Set 也是 Key 为 Int 类型，但是 Value 值是不能重复的特殊 Map。

7.1.2　Kotlin 集合类继承层次

下面是 Kotlin 中集合类接口的结构层次，如图 7-2 所示。

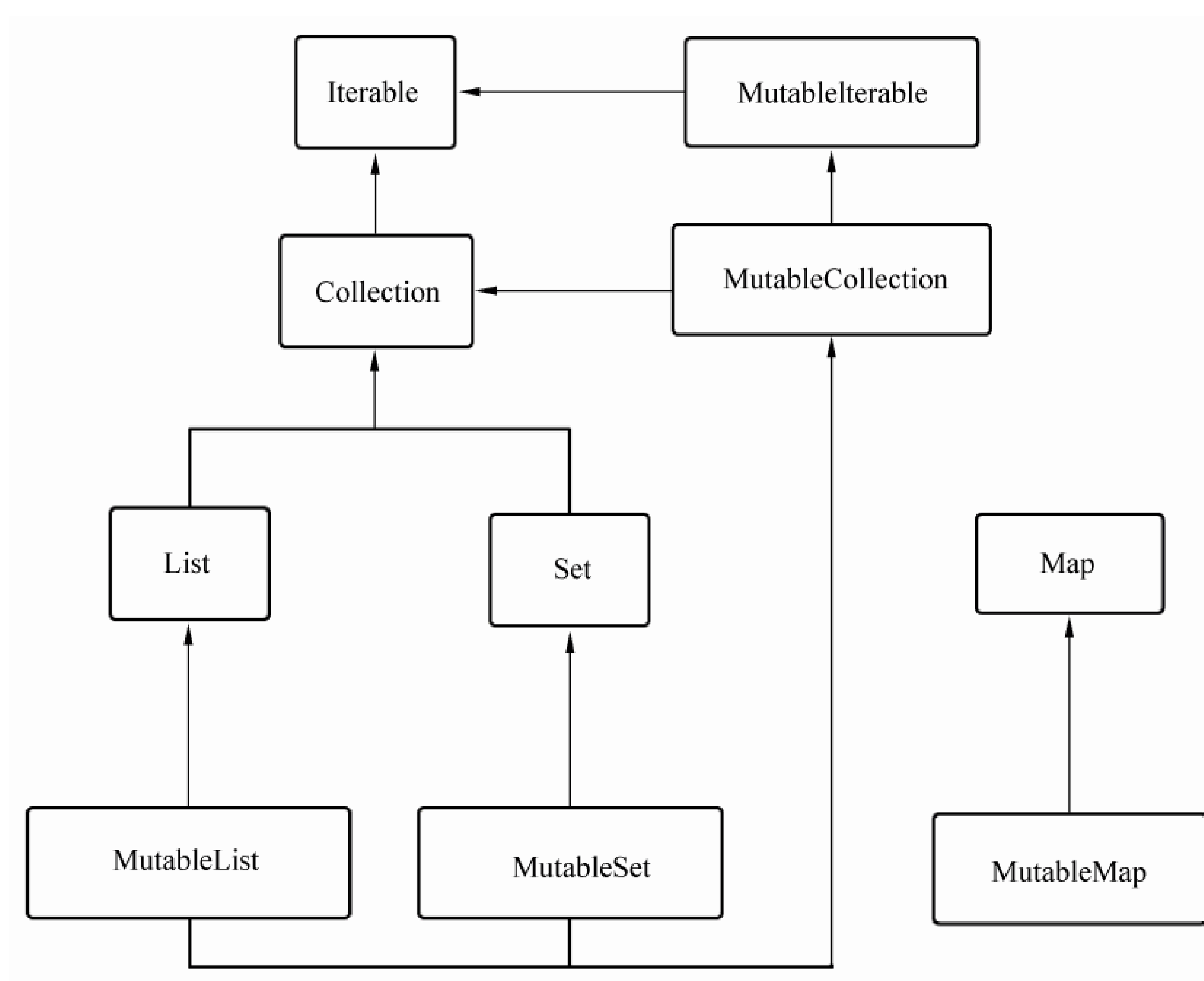


图 7-2　集合类接口结构层次

其中，各个接口说明如表 7-1 所示。

表 7-1　集合类接口说明

接　　口	功　　能
Iterable	父类。任何类继承这个接口就表示可以遍历序列的元素
MutableIterable	在迭代期间支持删除元素的迭代
Collection	List 和 Set 的父类接口。只读不可变
MutableCollection	支持添加和删除元素的 Collection。它提供写入的函数，如 add、remove 或 clear 等
List	最常用的集合，继承 Collection 接口，元素有序，只读不可变
MutableList	继承 List，支持添加和删除元素，除了拥有 List 中读数据的函数，还有 add、remove 或 clear 等写入数据的函数
Set	元素无重复、无序。继承 Collection 接口。只读不可变
MutableSet	继承 Set，支持添加和删除元素的 Set
Map	存储 K-V（键-值）对的集合。在 Map 映射表中 Key（键）是唯一的
MutableMap	支持添加和删除元素的 Map

7.2　不可变集合类

List 列表分为只读不可变的 List 和可变 MutableList（可写入、删除数据）。List 列表的类型层次结构如图 7-3 所示。

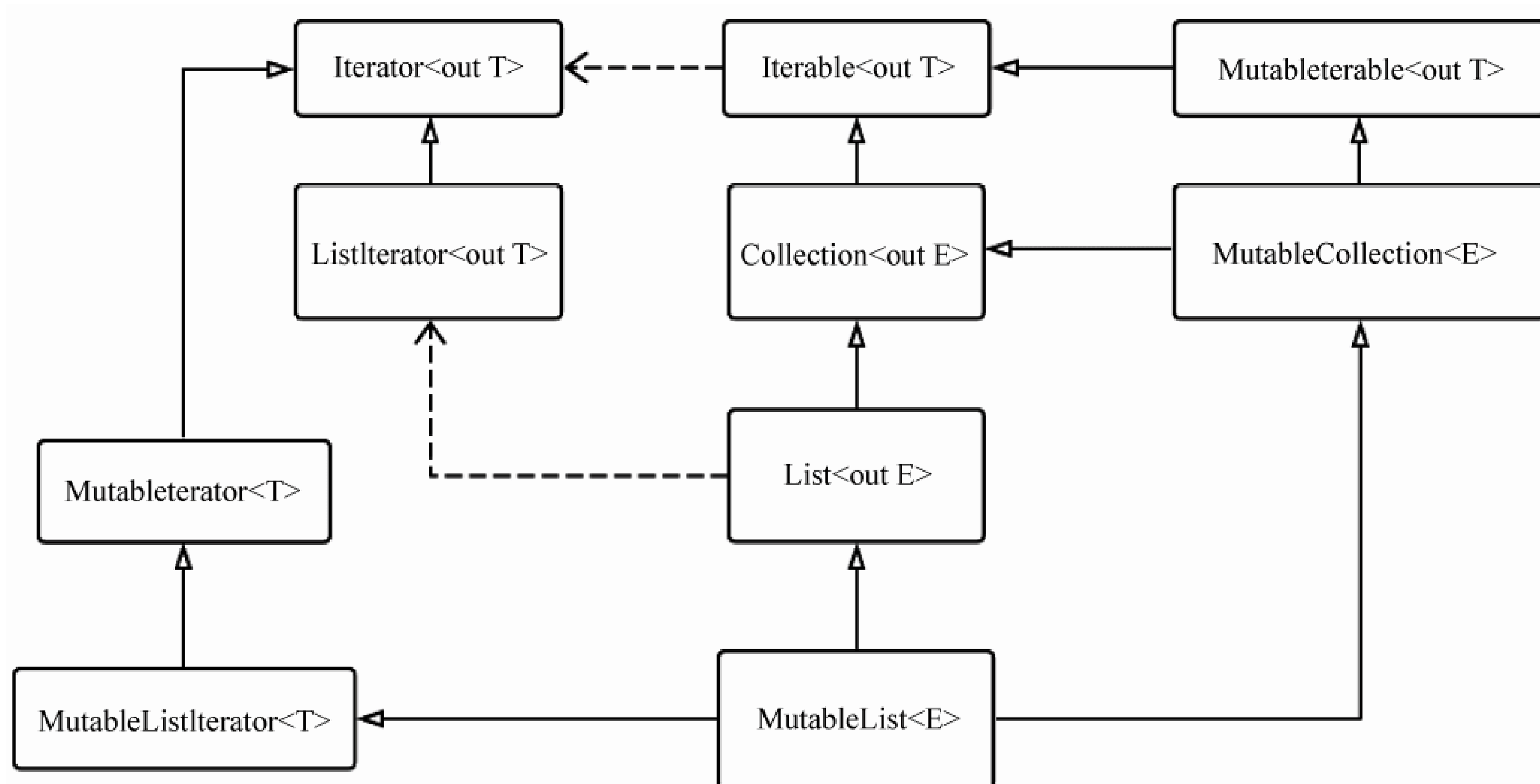


图 7-3　List 列表的类型层次结构

Set 集也分为不可变 Set 和可变 MutableSet（可写入、删除数据）。Set 集合的类型层次结构如图 7-4 所示。

Kotlin 中的 Map 与 List、Set 一样，Map 也分为只读 Map 和可变 MutableMap（可写入、删除数据）。Map 没有继承于 Collection 接口。Map 类型的层次结构如图 7-5 所示。

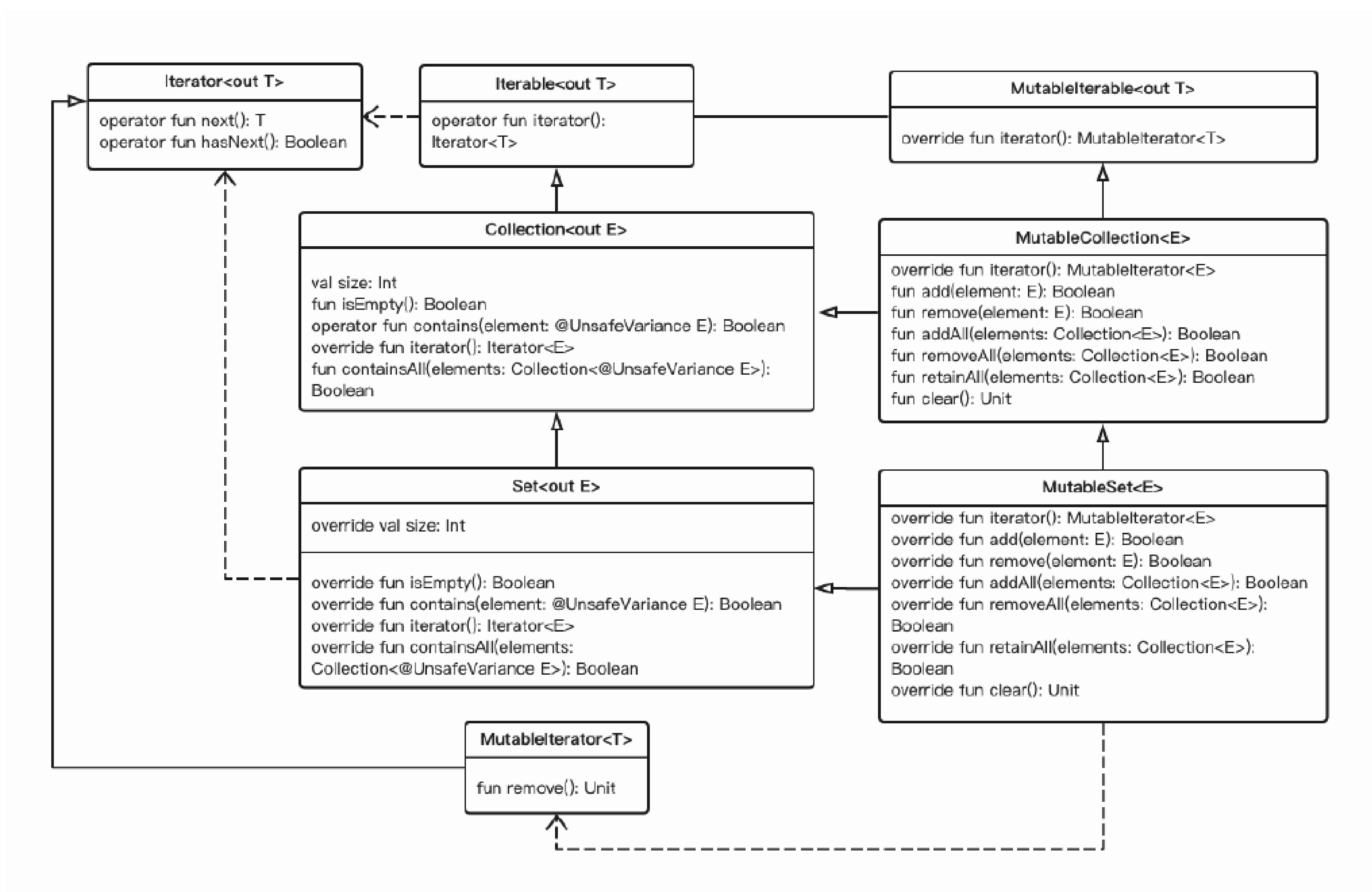


图 7-4　Set 集合的类型层次结构

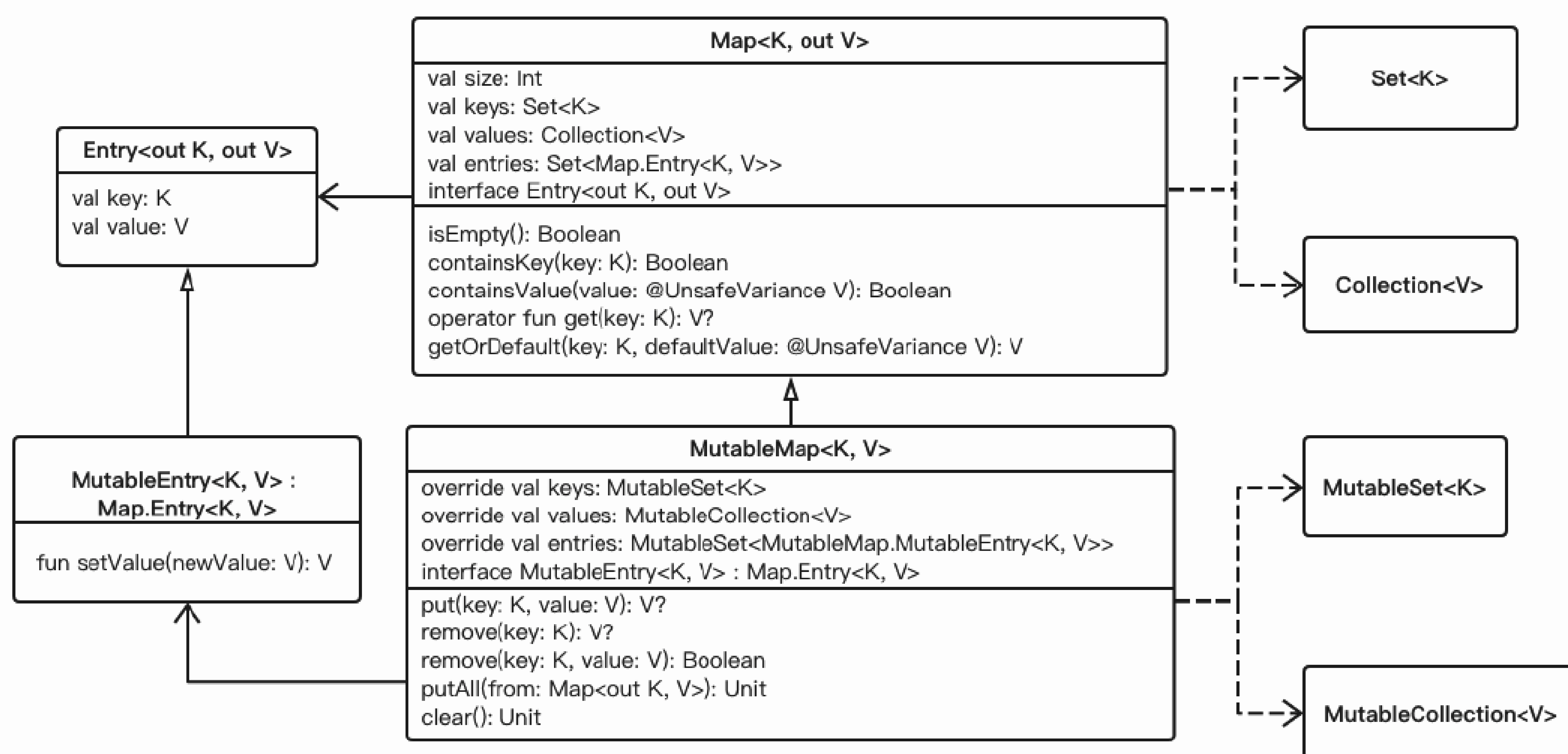


图 7-5　Map 类型的层次结构

下面，我们来创建集合类。

7.3　创建集合类

Kotlin 中分别使用 listOf()、setOf()、mapOf()函数创建不可变的 List 列表容器、Set 集

容器、Map 映射容器；使用 mutableListOf()、mutableSetOf()、mutableMapOf()函数来创建可变的 MutableList 列表容器、MutableSet 集容器、MutableMap 映射容器，分别如图 7-6 至图 7-8 所示。

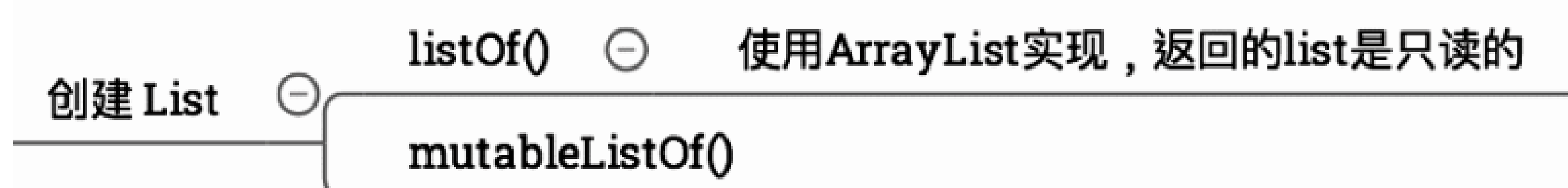


图 7-6　创建 List

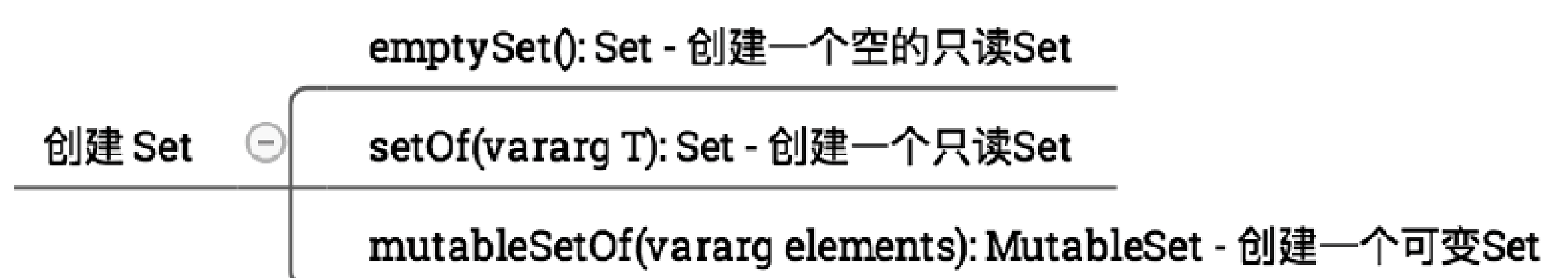


图 7-7　创建 Set

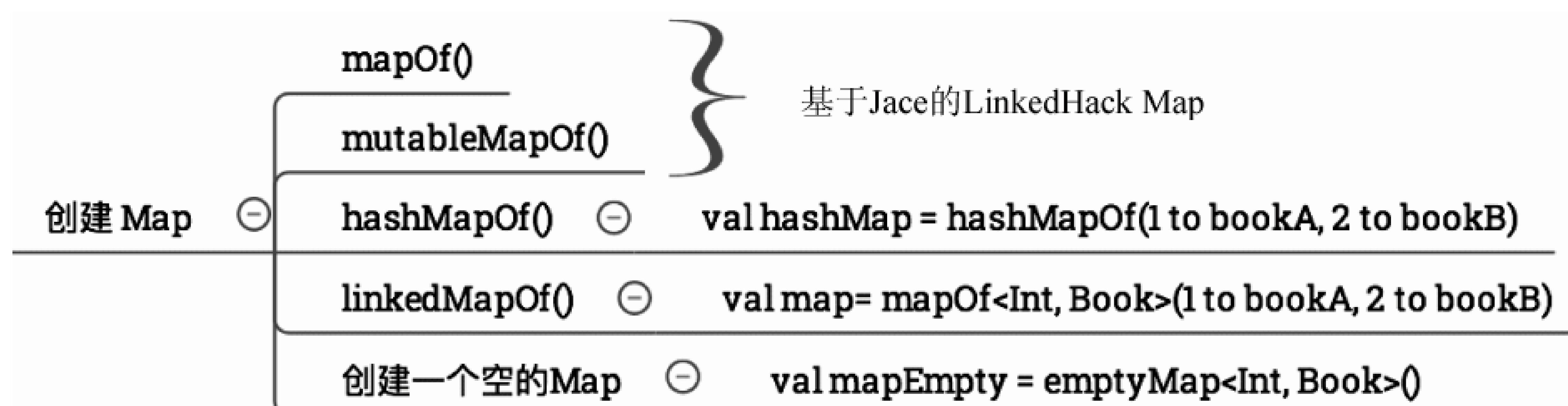


图 7-8　创建 Map

代码示例如下：

```
val list = listOf(1, 2, 3, 4, 5, 6, 7)              //创建不可变 List
val mutableList = mutableListOf("a", "b", "c")      //创建可变 MutableList

val set = setOf(1, 2, 3, 4, 5, 6, 7)                //创建不可变 Set
val mutableSet = mutableSetOf("a", "b", "c")        //创建可变 MutableSet

val map = mapOf(1 to "a", 2 to "b", 3 to "c")       //创建不可变 Map
val mutableMap = mutableMapOf(1 to"X",2 to"Y",3 to"Z") //创建可变 MutableMap
```

如果创建没有元素的空 List，使用 listOf()即可。不过这个时候，变量的类型不能省略，需要显式声明：

```
val emptyList: List<Int> = listOf()  //显式声明 List 的元素类型为 Int
val emptySet: Set<Int> = setOf()     //显式声明 Set 的元素类型为 Int
val emptyMap: Map<Int, String> = mapOf()//显式声明 Map 的元素类型为 Int, String 键值对
```

否则会报错：

```
>>> val list = listOf()                    //不指定类型，声明一个空 List 会报错
error: type inference failed: Not enough information to infer parameter T
in inline fun <T> listOf(): List<T>
```

```
Please specify it explicitly.
val list = listOf()
          ^
```

因为这里的 funlistOf():List 泛型参数 T 编译器无法推断出来。setOf()、mapOf()分析同理，不再赘述。

7.4　遍历集合中的元素

List、Set 类继承了 Iterable 接口，里面扩展了 forEach 函数来迭代遍历元素；同样，Map 接口中也扩展了 forEach 函数来迭代遍历元素。

```
list.forEach {       //List 中的 forEach
   println(it)
}
set.forEach {        //Set 中的 forEach
   println(it)
}

map.forEach {        //Map 中的 forEach
    println("K = ${it.key}, V = ${it.value}") //Map 里面的对象是 Map.Entry<K, V>
}
```

其中，forEach()函数签名如下：

```
public inline fun <T> Iterable<T>.forEach(action: (T) -> Unit): Unit
public inline fun <K, V> Map<out K, V>.forEach(action: (Map.Entry<K, V>)
-> Unit): Unit
```

我们看到，在 Iterable 和 Map 中，forEach 函数都是一个内联 inline 函数。

另外，如果我们想在迭代遍历元素的时候访问 index 下标，在 List 和 Set 中可以使用下面的 forEachIndexed 函数

```
list.forEachIndexed { index, value ->  //带下标 index 来遍历 List
   println("list index = ${index} , value = ${value}")
}

set.forEachIndexed { index, value ->   //带下标 index 来遍历 Set
   println("set index = ${index} , value = ${value}")
}
```

其中，第 1 个参数是 index，第 2 个参数是 value。这里的 forEachIndexed 函数签名如下：

```
public inline fun <T> Iterable<T>.forEachIndexed(action: (index: Int, T)
-> Unit): Unit
```

Map 的元素是 Entry 类型，由 entries 属性持有。

```
val entries: Set<Entry<K, V>>
```

这个 Entry 类型定义如下：

```
public interface Entry<out K, out V> {
       public val key: K    //键值对的 Key
```

```
        public val value: V        //键值对的 Value
    }
```

我们可以直接访问 entries 属性获取该 Map 中的所有键/值对的 Set。代码示例如下：

```
>>> val map = mapOf("x" to 1, "y" to 2, "z" to 3)  //声明并初始化一个 Map
>>> map
{x=1, y=2, z=3}
>>> map.entries                                  //访问 Map 对象的 entries 属性
[x=1, y=2, z=3]
```

这样就可以遍历这个 Entry 的 Set 了：

```
>>> map.entries.forEach({println("key="+ it.key + " value=" + it.value)})
                                                        //遍历 entries 中的元素
key=x value=1
key=y value=2
key=z value=3
```

7.5 映 射 函 数

使用 map 函数，可以把集合中的元素依次使用给定的转换函数进行映射操作，元素映射之后的新值会存入一个新的集合中，并返回这个新集合。这个过程可以用图 7-9 来说明。

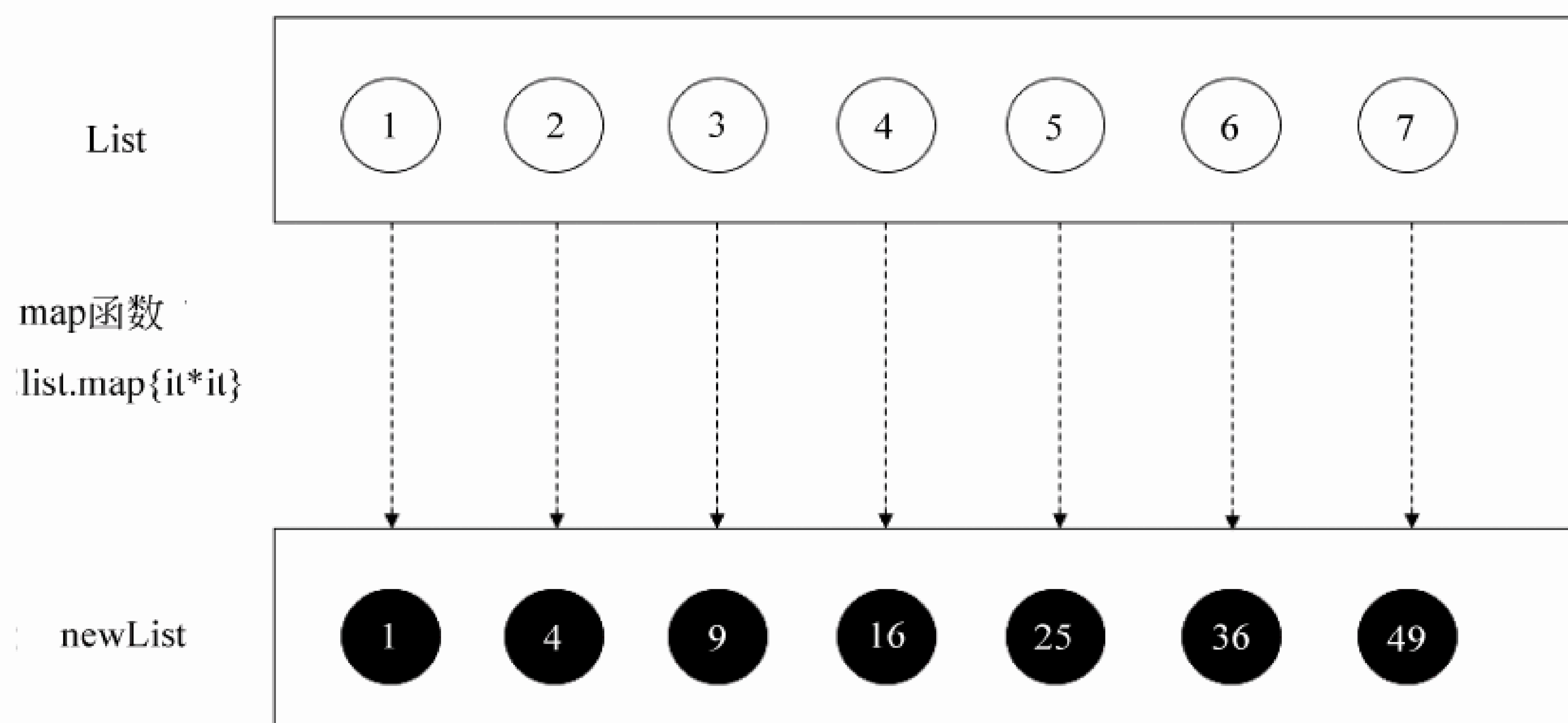


图 7-9　map 函数的映射图

在 List、Set 继承的 Iterable 接口和 Map 接口中，都提供了这个 map 函数。使用 map 函数的代码示例如下：

```
val list = listOf(1, 2, 3, 4, 5, 6, 7)          //声明并初始化一个 List
val set = setOf(1, 2, 3, 4, 5, 6, 7)            //声明并初始化一个 Set
val map = mapOf(1 to "a", 2 to "b", 3 to "c")  //声明并初始化一个 Map

list.map { it * it } //map 函数对每个元素进行乘方操作，返回 [1, 4, 9, 16, 25, 36, 49]
set.map{ it + 1 }   //map 函数对每个元素进行加 1 操作，返回[2, 3, 4, 5, 6, 7, 8]
map.map{ it.value + "$" } //map 函数对每个元素后加上字符$，返回[a$, b$, c$]
```

map 函数的签名如下：

```
public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R>
public inline fun <K, V, R> Map<out K, V>.map(transform: (Map.Entry<K, V>)
-> R): List<R>
```

这里的 R 类型是映射之后的数据类型，我们也可以传入一个 List：

```
val strlist = listOf("a", "b", "c")
strlist.map { it -> listOf(it + 1, it + 2, it + 3, it + 4) }
                                    //map 函数，每个元素 it 映射之后返回一个 List，这个
                                      List 中有 4 个元素，分别是 it+1,it+2, it+3,it+4
```

这个时候，返回值的类型将是 List，也就是一个 List 里面嵌套一个 List，上面代码的返回结果如下：

```
[[a1, a2, a3, a4], [b1, b2, b3, b4], [c1, c2, c3, c4]] //嵌套 List
```

Kotlin 中还提供了一个 flatten()函数，效果是把嵌套的 List 结构“平铺”，变成一层的结构，代码示例如下：

```
strlist.map { it -> listOf(it + 1, it + 2, it + 3, it + 4) }.flatten()
                              //“平铺”函数，把嵌套在 List 中的元素“平铺”成一层 List
```

输出如下：

```
[a1, a2, a3, a4, b1, b2, b3, b4, c1, c2, c3, c4]
```

flatMap 函数是 map 和 flat 两个函数的“复合逻辑”，代码示例如下：

```
strlist.flatMap { it -> listOf(it + 1, it + 2, it + 3, it + 4) }
```

同样输出如下：

```
[a1,a2,a3,a4,b1,b2,b3, b4,c1,c2,c3,c4]
```

7.6　过滤函数

在第 5 章中，我们已经讲过了 filter()函数，这里再举一个代码示例。首先，有一个 Student 对象，我们使用数据类来声明如下：

```
data class Student(var id: Long, var name: String, var age: Int, var score:
Int){                                          //声明 Student 数据类
    override fun toString(): String {  //覆盖写 toString()函数
        return "Student(id=$id, name='$name', age=$age, score=$score)"
    }
}
```

为了方便读者看到打印信息，重写了 toString()函数。然后创建一个持有 Student 对象的 List：

```
val studentList = listOf(                  //创建一个持有 Student 对象 List
    Student(1, "Jack", 18, 90),
    Student(2, "Rose", 17, 80),
```

```
    Student(3, "Alice", 16, 70)
)
```

这个时候，如果我们想要过滤出年龄大于等于 18 岁的学生，代码可以这样写：

```
studentList.filter { it.age >= 18 } //过滤出 studentList 中年龄大于等于 18 岁的
                                      Student 对象
```

输出如下：

```
[Student(id=1, name='Jack', age=18, score=90)]
```

如果想要过滤出分数小于 80 分的学生，代码如下：

```
studentList.filter { it.score < 80 } //过滤出 studentList 中分数小于 80 分的
                                       Student 对象
```

输出如下：

```
[Student(id=3, name='Alice', age=16, score=70)]
```

另外，如果想要通过访问下标来过滤，可以使用 filterIndexed()函数：

```
val list = listOf(1, 2, 3, 4, 5, 6, 7)
list.filterIndexed { index, it -> index % 2 == 0 && it > 3 }
                                       //带下标过滤 List 中的元素，返回[5, 7]
```

filterIndexed()函数签名如下：

```
public inline fun <T> Iterable<T>.filterIndexed(predicate: (index: Int, T)
-> Boolean): List<T>
```

7.7 排 序 函 数

Kotlin 集合类中提供了倒序排列集合元素的函数 reversed()，代码示例如下：

```
val list = listOf(1, 2, 3, 4, 5, 6, 7)
val set = setOf(1,3,2)

list.reversed()      //倒序函数，返回[7, 6, 5, 4, 3, 2, 1]
set.reversed()       //倒序函数，返回[2, 3, 1]
```

这个 Iterable 的扩展函数 reversed()是直接调用的 java.util.Collections.reverse()方法。其相关代码如下：

```
public fun <T> Iterable<T>.reversed(): List<T> {
    if (this is Collection && size <= 1) return toList()
    val list = toMutableList()
    list.reverse() //调用 Java 中 List 类型的 reverse()方法
    return list
}

public fun <T> MutableList<T>.reverse(): Unit {
    java.util.Collections.reverse(this)
}
```

升序排序函数是 sorted()，实例代码如下：

```
>>> list.sorted()
[1, 2, 3, 4, 5, 6, 7]
>>> set.sorted()
[1, 2, 3]
```

Kotlin 中的这个 sorted()函数也是直接调用 Java 的 API 来实现的，相关代码如下：

```
public fun <T : Comparable<T>> Iterable<T>.sorted(): List<T> {
    if (this is Collection) {
        if (size <= 1) return this.toList()
        @Suppress("UNCHECKED CAST")
        return(toTypedArray<Comparable<T>>()as Array<T>).apply { sort()}.asList()
    }
    return toMutableList().apply { sort() }
}
```

其背后调用的是 Java.util.Arrays.sort()方法：

```
public fun <T> Array<out T>.sort(): Unit {
    if (size > 1) java.util.Arrays.sort(this)//调用 Java 中 Arrays 类型的 sort 方法
}
```

7.8　元 素 去 重

如果我们想对一个 List 列表进行元素去重，可以直接调用 distinct()函数：

```
val dupList = listOf(1, 1, 2, 2, 3, 3, 3)
dupList.distinct()          //去重函数，返回 [1, 2, 3]
```

Kotlin 的集合类中还提供了许多功能丰富的 API，此处不再一一介绍。更多内容可以参考官方 API 文档：

http://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html。

7.9　本 章 小 结

本章我们介绍了 Kotlin 标准库中的集合类 List、Set、Map，以及它们扩展的丰富的操作函数，这些函数使得我们使用这些集合类更容易。集合类持有的是对象，而怎样放入正确的对象类型则是我们写代码过程中需要注意的。第 8 章中我们将学习泛型。

第 8 章　泛　　型

通常情况的类和函数，我们只需要使用具体的类型即可：要么是基本类型，要么是自定义的类。但是在集合类的场景下，我们通常需要编写可以应用于多种类型的代码，最简单的做法是针对每一种类型均写一套刻板的代码。但这样做使代码的复用率很低，抽象也没有做好。那么我们能不能把“类型”也抽象成参数呢？答案是当然可以。

Java 5 中引入的泛型机制实现了“参数化类型”（Parameterized Type）。参数化类型，顾名思义就是将类型由原来的具体类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式，我们称之为类型参数，然后在使用时传入具体的类型（类型实参）。

我们知道，在数学中泛函是以函数为自变量的函数。类似的，编程中的泛型就是以类型为变量的类型，即参数化类型。这样的变量参数就叫类型参数（Type Parameters）。

本章我们来一起学习 Kotlin 泛型的相关知识。

8.1　为何引入泛型

《Java 编程思想 》（第 4 版）中提到：有许多原因促成了泛型的出现，而最引人注意的一个原因，就是为了创建容器类（集合类）。

集合类可以说是我们在写代码的过程中最常用的类之一。我们先来看下没有泛型之前，集合类是怎样持有对象的。在 Java 中，Object 类是所有类的根类。为了集合类的通用性，把元素的类型定义为 Object，当放入具体的类型时，再进行相应的强制类型转换。

下面是一个示例代码：

```
class RawArrayList {
    public int length = 0;
    private Object[] elements;          //把元素的类型定义为 Object

    public RawArrayList(int length) {   //构造函数
        this.length = length;
        this.elements = new Object[length];//创建一个长度为 length 的 Object 数组
    }

    public Object get(int index) {      //get 方法
        return elements[index];
    }

    public void add(int index,Object element){//给下标位置为 index 的元素赋值为 element
         elements[index] = element;
    }
}
```

一个简单的测试代码如下：

```
public class RawTypeDemo {

    public static void main(String[] args) {
        RawArrayList rawArrayList = new RawArrayList(4);
                        //创建并初始化一个持有元素长度为 4 的 RawArrayList 对象
        rawArrayList.add(0, "a");                   //给下标 0 的位置赋值为"a"
        rawArrayList.add(1, "b");                   //给下标 1 的位置赋值为"b"
        System.out.println(rawArrayList);

        String a = (String)rawArrayList.get(0); //返回下标为 0 的元素
        System.out.println(a);

        String b = (String)rawArrayList.get(1); //返回下标为 1 的元素
        System.out.println(b);

        rawArrayList.add(2, 200);
        rawArrayList.add(3, 300);
        System.out.println(rawArrayList);

        int c = (int)rawArrayList.get(2);
        int d = (int)rawArrayList.get(3);
        System.out.println(c);
        System.out.println(d);

        String x = (String)rawArrayList.get(2); //Exception in thread "main"
         java.lang.ClassCastException: java.lang.Integer cannot be cast to
         java.lang.String
        System.out.println(x);

    }

}
```

可以看出，在使用原生态类型（raw type）实现的集合类中，我们使用的是 Object[]数组。这种实现方式存在的问题有两个：

- 向集合中添加对象元素的时候，没有对元素的类型进行检查。也就是说，我们向集合中添加任意对象，编译器都不会报错。
- 当我们从集合中获取一个值的时候，不能都使用 Object 类型，需要进行强制类型转换。而这个转换过程由于在添加元素的时候没有做任何的类型的限制与检查，所以容易出错。例如上面代码中的：

```
String x = (String)rawArrayList.get(2); //Exception in thread "main"
java.lang.ClassCastException: java.lang.Integer cannot be cast to
java.lang.String
```

对于这行代码，编译时不会报错，但是运行时会抛出类型转换错误。能不能让编译器来解决这样样板化的类型转换代码呢？能否在我们向 rawArrayList 添加元素的时候

```
rawArrayList.add(0, "a");
```

就限定其元素类型只能为 String，然后在后面获取元素的时候，自动强制转型为 String 呢？

```
String a = (String)rawArrayList.get(0);
```

我们将这个元素类型 String 的信息存放到一个“类型参数”中，然后在编译器层面引入相应的类型检查和自动转换机制，这样就可以解决类型安全使用的问题了。这也正是引入泛型的基本思想。

泛型最主要的优点就是让编译器追踪参数类型，执行类型检查和类型转换。因为由编译器来保证类型转换不会失败，如果依赖程序员自己去追踪对象类型和执行转换，那么运行时产生的错误将很难去定位和调试。有了泛型，编译器就可以帮助我们执行大量的类型检查，并且可以检测出更多的编译时错误。在这一点上，泛型与第 3 章中所讲到的“可空类型”实现的空指针安全，在思想上有着异曲同工之妙。

8.2 在类、接口和函数上使用泛型

泛型类、泛型接口和泛型方法具备可重用性、类型安全和高效等优点。在集合类 API 中大量地使用了泛型。在 Java 中我们可以为类、接口和方法分别定义泛型参数，在 Kotlin 中也同样支持。本节我们分别介绍 Kotlin 中的泛型接口、泛型类和泛型函数。

8.2.1 泛型接口

下面先举一个简单的 Kotlin 泛型接口的例子。

```
interface Generator<T> {        //类型参数放在接口名称后面：<T>
   operator fun next(): T       //接口函数中直接使用类型 T
}
```

测试代码如下：

```
fun testGenerator() {
   val gen = object : Generator<Int> { //对象表达式
      override fun next(): Int {
         return Random().nextInt(10)
      }
   }
   println(gen.next())
}
```

这里我们使用 object 关键字来声明一个 Generator 实现类，并在 Lambda 表达式中实现了 next()函数。

Kotlin 中 Map 和 MutableMap 接口的定义也是一个典型的泛型接口的例子。

```
public interface Map<K, out V> { Map 接口的泛型声明 <K, out V >，关于 out 参数，我们在后面会讲解
    ...
   public fun containsKey(key: K): Boolean
   public fun containsValue(value: @UnsafeVariance V): Boolean
   public operator fun get(key: K): V?
   ...
```

```
    public val keys: Set<K>
    public val values: Collection<V>
    public val entries: Set<Map.Entry<K, V>>
}

public interface MutableMap<K, V> : Map<K, V> {
    public fun put(key: K, value: V): V?
    public fun remove(key: K): V?
    public fun putAll(from: Map<out K, V>): Unit
    ...
}
```

例如，使用 mutableMapOf()函数来实例化一个可变 Map：

```
>>> val map = mutableMapOf<Int,String>(1 to "a", 2 to "b", 3 to "c")
>>> map
{1=a, 2=b, 3=c}
```

其中，mutableMapOf()函数签名如下：

```
fun <K, V> mutableMapOf(vararg pairs: Pair<K, V>): MutableMap<K, V>
```

这里的类型参数 K 和 V 在泛型类型被实例化和使用时，将被实际的类型参数所替代。在 mutableMapOf()函数中，放置 K、V 的位置被具体的 Int 和 String 类型所替代。

泛型可以用来限制集合类持有的对象类型，这样使得类型更加安全。当我们在一个集合类里面放入了错误类型的对象时，编译器就会报错：

```
>>> map.put("5","e")
error: type mismatch: inferred type is String but Int was expected
map.put("5","e")
        ^
```

Kotlin 中有类型推断的功能，有些类型参数可以直接省略不写。mutableMapOf()函数后面的类型参数可以省掉不写：

```
>>> val map = mutableMapOf(1 to "a", 2 to "b", 3 to "c")
>>> map
{1=a, 2=b, 3=c}
```

8.2.2　泛型类

我们直接声明一个带类型参数的 Container 类，代码如下：

```
class Container<K, V>(var key: K, var value: V)
```

为了方便测试，我们重写 toString()函数，代码如下：

```
class Container<K, V>(var key: K, var value: V){ //在类名后面声明泛型参数<K,
                                                   V>，多个泛型使用逗号隔开
    override fun toString(): String {
        return "Container(key=$key, value=$value)"
    }
}
```

测试代码如下：

```
fun testContainer() {
```

```
    val container= Container<Int, String>(1, "A") // <K, V> 被具体化为<Int, String>
    println(container) //输出: container = Container(key=1, value=A)
}
```

8.2.3 泛型函数

在泛型接口和泛型类中，我们都在类名和接口名后面声明了泛型参数。而实际上也可以直接在类或接口中的函数声明泛型参数或者在包级函数中直接声明泛型参数。代码示例如下：

```
class GenericClass {
    fun <T> console(t: T) {                              //类中的泛型函数
        println(t)
    }
}
interface GenericInterface {
    fun <T> console(t: T)                                //接口中的泛型函数
}
fun <T : Comparable<T>> gt(x: T, y: T): Boolean {  //包中的泛型函数
    return x > y
}
```

8.3 类 型 上 界

在上面的例子中，gt(x:T, y:T)函数的签名中有个 T:Comparable<T>：

```
fun <T : Comparable<T>> gt(x: T, y: T): Boolean //T 的类型上界是 Comparable<T>
```

这里的 T:Comparable，表示 Comparable 是类型 T 的上界。也就是告诉编译器，类型参数 T 代表的都是实现了 Comparable 接口的类，这样等于告诉编译器它们都实现了 CompareTo 方法。如果没有在这个类型上界进行声明，就无法直接使用 CompareTo “>”操作符。也就是说，下面的代码编译不通过。

```
fun <T> gt(x: T, y: T): Boolean {
    return x > y //编译不通过
}
```

8.4 协变与逆变

我们先来看一个问题场景。首先有下面存在父子关系的类型：

```
open class Food
open class Fruit : Food()    //Fruit 继承 Food
class Apple : Fruit()        //Apple 继承 Fruit
class Banana : Fruit()       //Banana 继承 Fruit
class Grape : Fruit()        //Grape 继承 Fruit
```

然后有下面两个函数：

```
object GenericTypeDemo {
    fun addFruit(fruit: MutableList<Fruit>) {
    }

    fun getFruit(fruit: MutableList<Fruit>) {
    }
}
```

这个时候可以这样调用上面的两个函数：

```
val fruits: MutableList<Fruit> = mutableListOf(Fruit(), Fruit(), Fruit())
GenericTypeDemo.addFruit(fruits)
GenericTypeDemo.getFruit(fruits)
```

现在又有一个存放 Apple 的 List：

```
val apples: MutableList<Apple> = mutableListOf(Apple(), Apple(), Apple())
```

由于 Kotlin 中的泛型与 Java 一样是非协变的，下面的调用是编译不通过的：

```
GenericTypeDemo.addFruit(apples) //type mismatch
GenericTypeDemo.getFruit(apples) //type mismatch
```

如果没有协变，那么我们不得不再添加两个函数：

```
object GenericTypeDemo {

    fun addFruit(fruit: MutableList<Fruit>) {
    }

    fun getFruit(fruit: MutableList<Fruit>) {
    }

    fun addApple(apple: MutableList<Apple>) {
    }

    fun getApple(apple: MutableList<Apple>) {
    }

}
```

我们一眼就能看出，这是重复的样板代码。那么能不能让 MutableList 成为 MutableList 的父类型呢？Java 泛型中引入了类型通配符的概念来解决这个问题。Java 泛型的通配符有两种形式：

- 子类型上界限定符?extendsT 指定类型参数的上限（该类型必须是类型 T 或者它的子类型）。也就是说 MutableList<?extends Fruit>是 MutableList 的父类型。Kotlin 中使用 MutableList 来表示。
- 超类型下界限定符?superT 指定类型参数的下限（该类型必须是类型 T 或者它的父类型）。也就是说 MutableList<?super Fruit>是 MutableList 的父类型。Kotlin 中使用 MutableList 来表示。

这里的问号“?”，称之为类型通配符（Type Wildcard）。通配符在类型系统中具有重要的意义，它们为一个泛型类所指定的类型集合提供了一个有用的类型范围。

Number 类型（简记为 F）是 Integer 类型（简记为 C）的父类型，我们把这种父子类型关系简记为 C=>F（C 继承 F）；而 List,List 代表的泛型类型信息分别简记为 f(F),f(C)。

那么我们可以这样描述协变和逆变：

- 当 C =>F 时，如果有 f(C) => f(F)，那么 f 叫做协变；
- 当 C => F 时，如果有 f(F) => f(C)，那么 f 叫做逆变。如果上面两种关系都不成立则叫做不变。

协变与逆变可以用图 8-1 来简单说明。

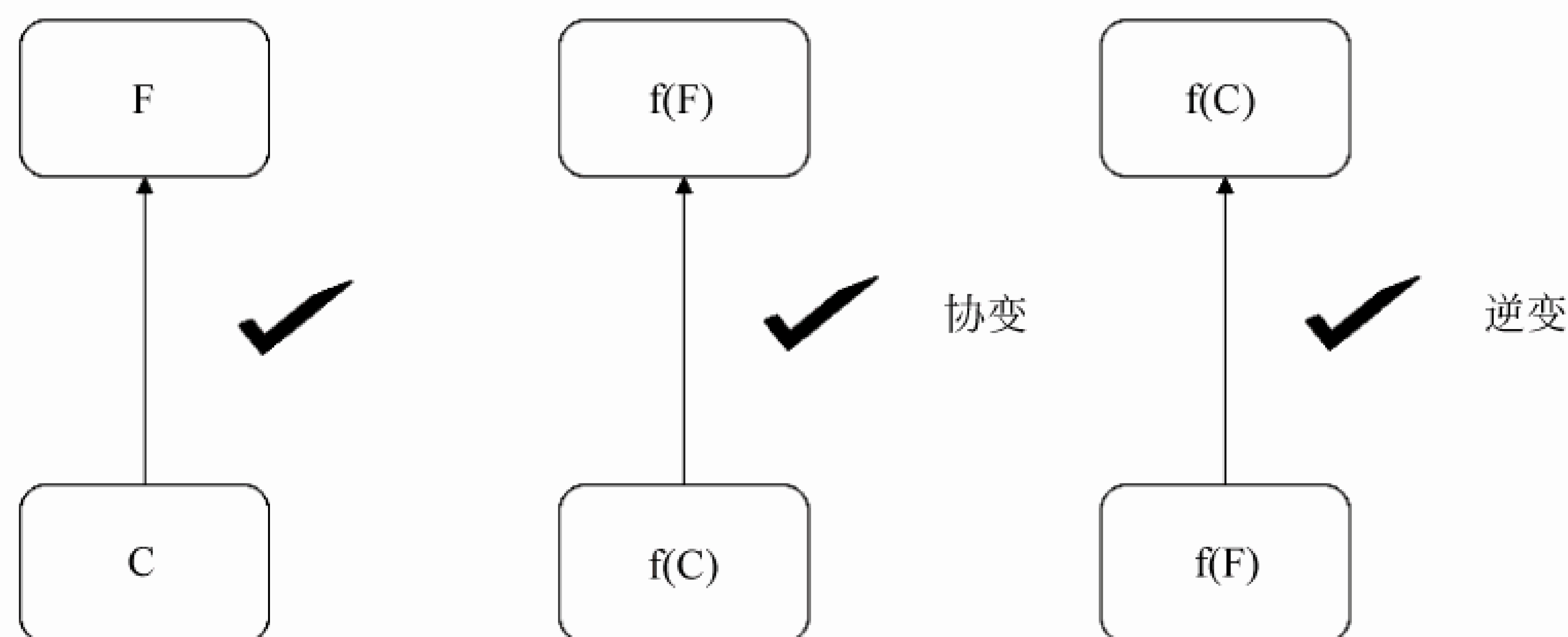


图 8-1　协变与逆变

协变和逆协变都是类型安全的。

8.4.1　协变

Java 中的数组是协变的，下面的代码是可以正确编译运行的：

```
Integer[] ints = new Integer[3];
ints[0] = 0;
ints[1] = 1;
ints[2] = 2;
Number[] numbers = new Number[3];
numbers = ints;     //数组是协变的，可以正确赋值
for (Number n : numbers) {
    System.out.println(n);
}
```

在 Java 中，因为 Integer 是 Number 的子类型，数组类型 Integer[]也是 Number[]的子类型，因此在任何需要 Number[]值的地方都可以提供一个 Integer[]值。Java 中数组协变的意思可以用图 8-2 来简单说明。

Java 中的泛型是非协变的，如图 8-3 所示。

也就是说，List<Integer>不是 List<Number>的子类型，在要求 List<Number>的位置提供 List<Integer>会提示类型错误。下面的代码，编译器是会直接报错的：

```
List<Integer> integerList = new ArrayList<>();
integerList.add(0);
integerList.add(1);
integerList.add(2);
List<Number> numberList = new ArrayList<>();
```

```
numberList = integerList; //编译错误：类型不兼容
```

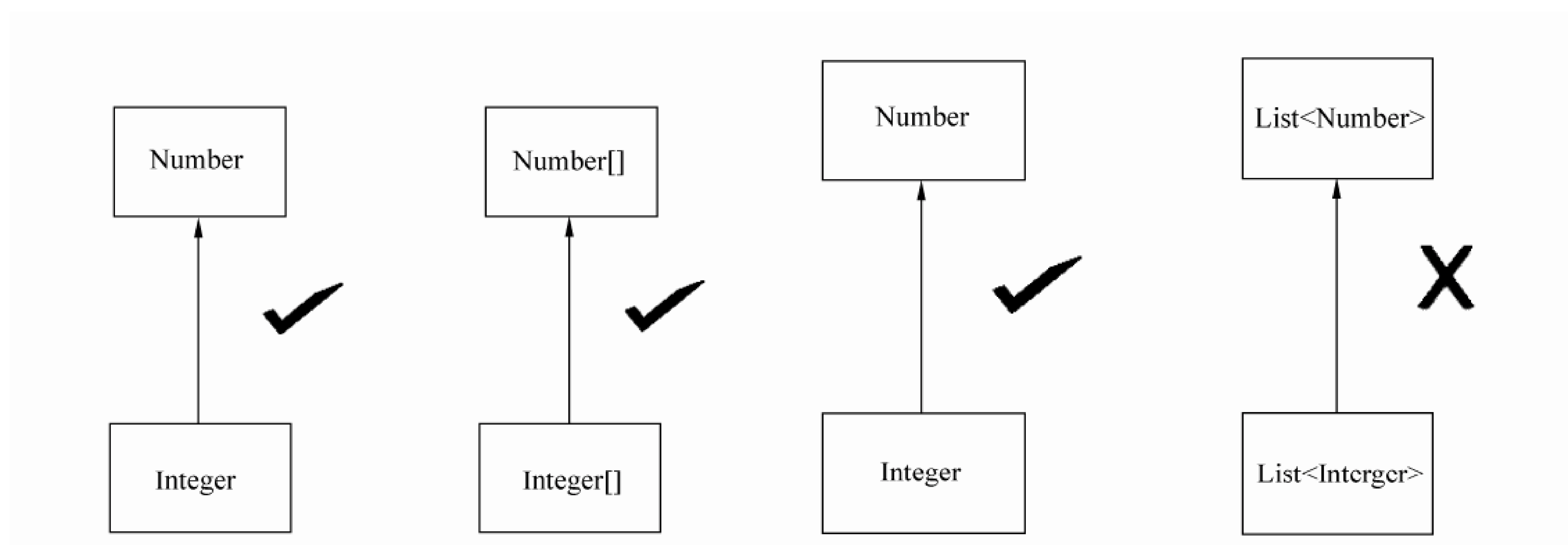


图 8-2　Java 中的数组是协变的　　　　图 8-3　Java 中的泛型是非协变的

编译器报错提示如图 8-4 所示。

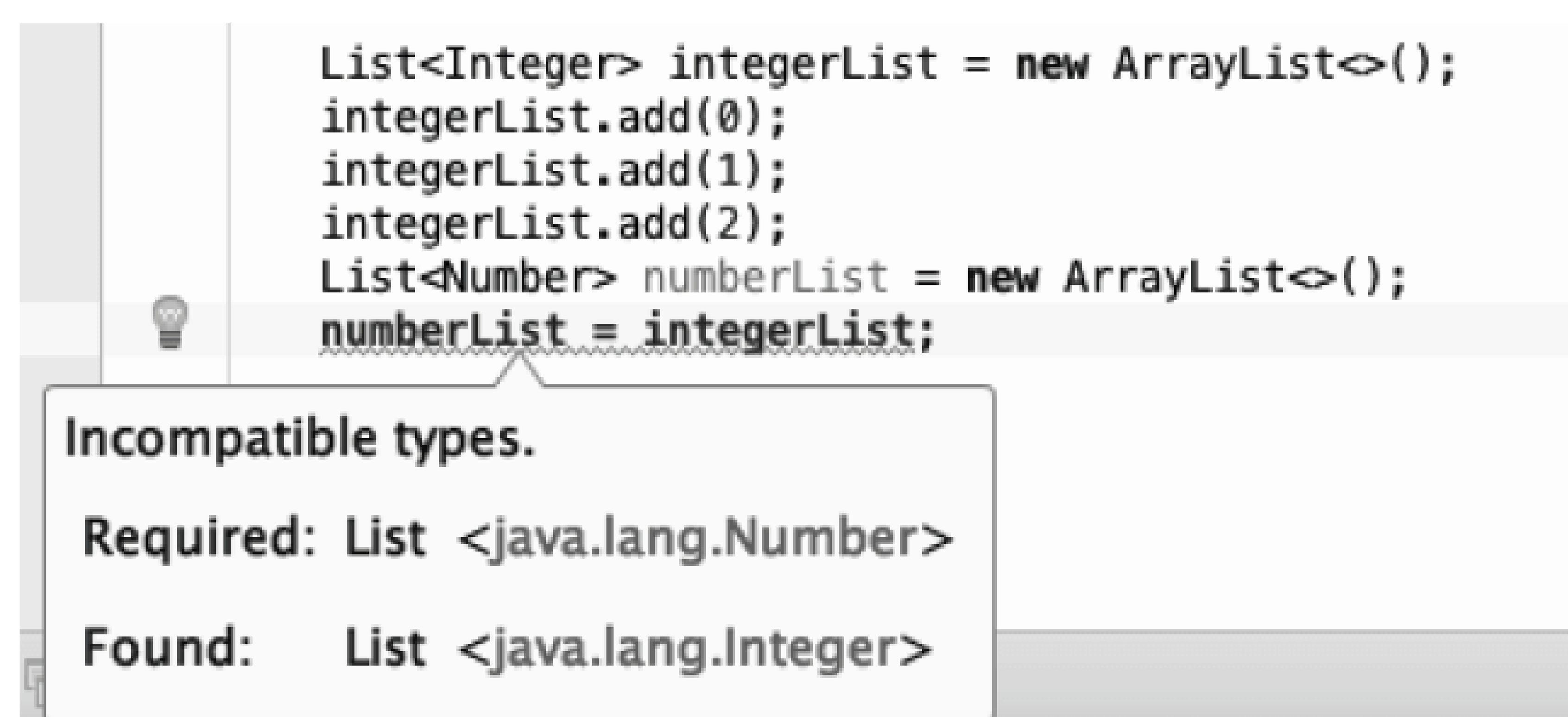


图 8-4　编译错误：类型不兼容

Java 中泛型和数组的不同行为的确引起了许多混乱，就算我们使用通配符这样写：

```
List<? extends Number> list = new ArrayList<Number>();
list.add(new Integer(1)); //error
```

仍然是报错的，如图 8-5 所示。

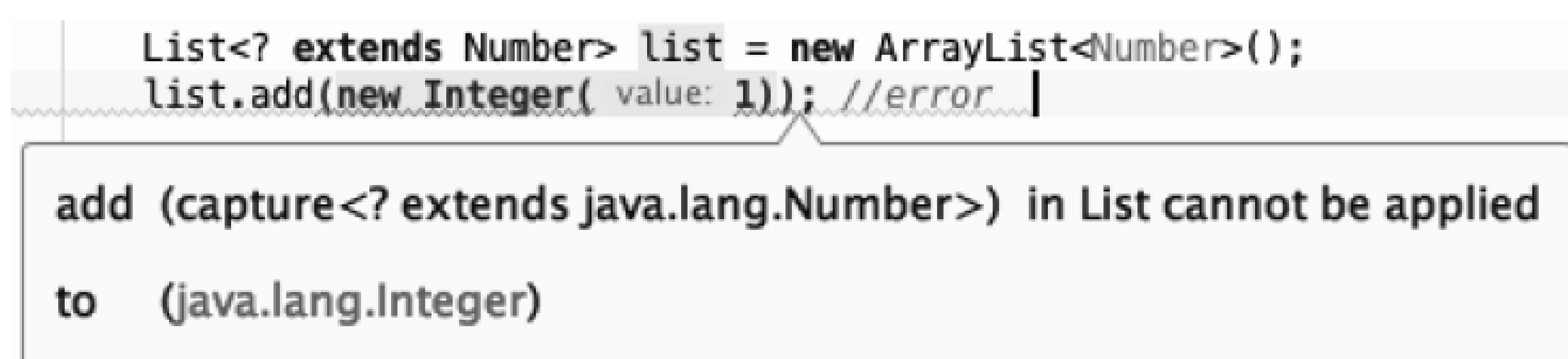


图 8-5　类型不兼容的报错信息

这通常会让我们感到困惑：为什么 Number 的对象可以由 Integer 实例化，而 ArrayList<Number>的对象却不能由 ArrayList<Integer>实例化？list 中的<? extends Number>声明其元素是 Number 或 Number 的派生类，为什么不能 add Integer？为了弄清楚这些问题，我们需要了解 Java 中的逆变和协变及泛型中通配符的用法。

```
List<? extends Number> list = new ArrayList<>();
```

这里的子类型 C 就是 Number 类及其子类（如 Number、Integer、Float 等），表示的是 Number 类或其子类。父类 F 就是上界通配符?extends Number。

协变的意义如图 8-6 所示。

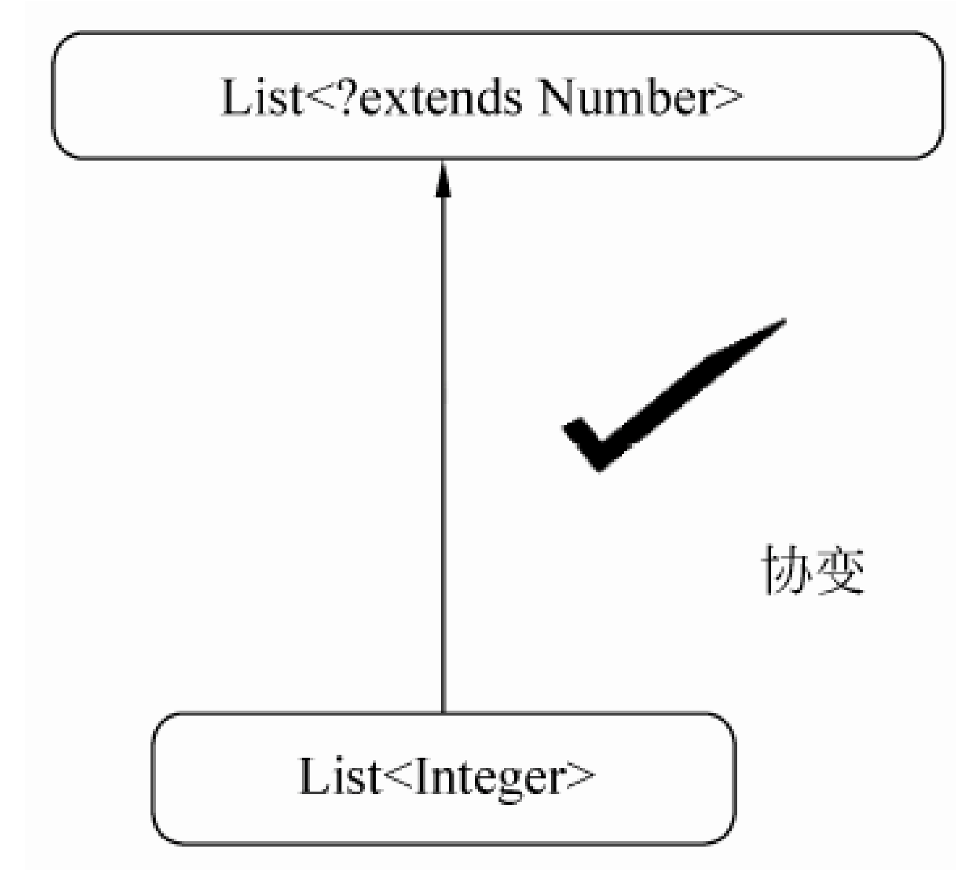


图 8-6　协变

代码示例如下：

```
List<? extends Number> list1 = new ArrayList<Integer>();
List<? extends Number> list2 = new ArrayList<Float>();
```

但是这里不能向 list1、list2 添加除 null 以外的任意对象。

```
list1.add(null);                 //ok
list2.add(null);                 //ok
list1.add(new Integer(1));       //error
list2.add(new Float(1.1f));      //error
```

List<Integer>可以添加 Integer 及其子类；List<Float>可以添加 Float 及其子类；List<Integer>、List<Float>等都是 List<?extends Number>的子类型。

现在问题来了，如果能将 Float 的子类添加到 List<?extends Number>中，也能将 Integer 的子类添加到 List<?extends Number>中，那么 List<?extends Number>里面将会持有各种 Number 子类型的对象（如 Byte、Integer、Float、Double 等）。而这个时候，当我们再使用这个 List 的时候，元素的类型就会混乱，我们不知道哪个元素是 Integer 或者 Float。Java 为了保护其类型一致，禁止向 List<? extends Number>添加任意 Number 子类型的对象，不过可以添加空对象 null，如图 8-7 所示。

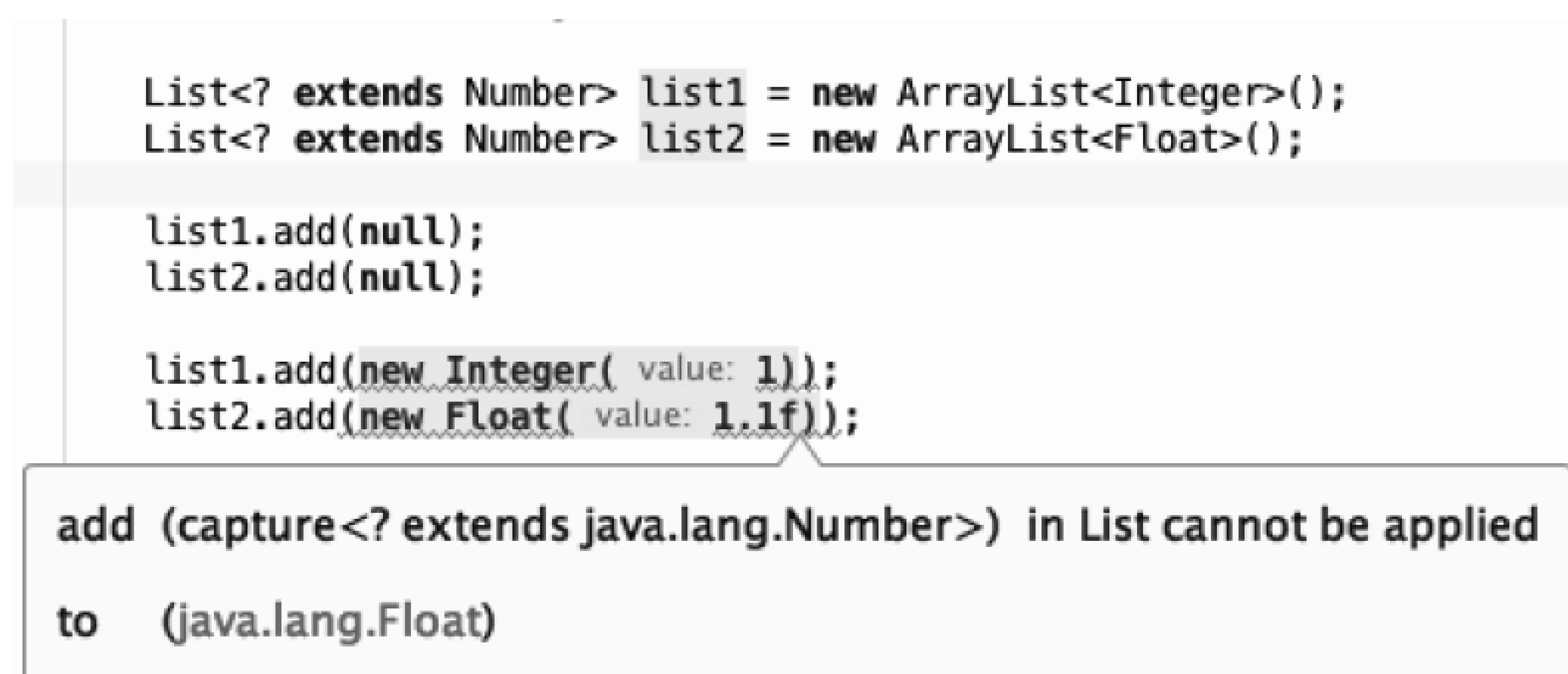


图 8-7　禁止向 List<? extends Number>添加任意 Number 子类型的对象

8.4.2　逆变

我们先用一段代码举例：

```
List<? super Number> list = new ArrayList<Object>();
```

这里的子类型 C 是“? super Number”，父类型 F 是 Number 的父类型（如 Object 类）。逆变的意义如图 8-8 所示。

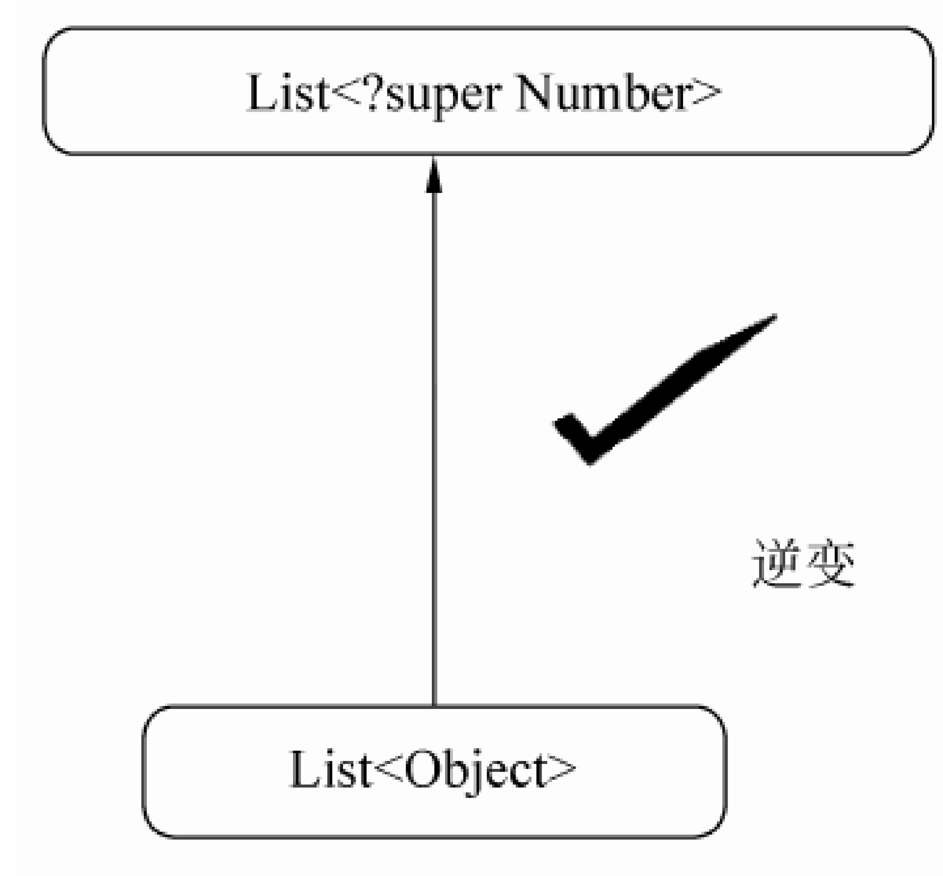


图 8-8　逆变

代码示例如下：

```
List<? super Number> list3 = new ArrayList<Number>();
                                   //Java 中的<? super Number>通配符
List<? super Number> list4 = new ArrayList<Object>();
list3.add(new Integer(3));         //可以添加 Integer 类型的元素
list4.add(new Integer(4));
```

在逆变类型中，我们可以向其中添加元素。例如，可以向 List<? super Number>list4 变量中添加 Number 及其子类对象。

8.4.3　PECS

现在问题来了：我们什么时候用 extends，什么时候用 super 呢？

Effective Java 给出了答案：PECS（Producer-Extends,Consumer-Super）。

Naftalin 与 Wadler 将 PECS 称为 Get and Put Principle。

在 java.util.Collections 的 copy()方法中（JDK1.7）完美地诠释了 PECS，代码如下：

```
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
    int srcSize = src.size();
    if (srcSize > dest.size())
       throw new IndexOutOfBoundsException("Source does not fit in dest");

     if (srcSize < COPY_THRESHOLD ||
        (src instanceof RandomAccess && dest instanceof RandomAccess)) {
        for (int i=0; i<srcSize; i++)
           dest.set(i, src.get(i));
```

```
    } else {
        ListIterator<? super T> di=dest.listIterator();//in T，写入 dest 数据
        ListIterator<? extends T> si=src.listIterator();//out T，读取 src 数据
        for (int i=0; i<srcSize; i++) {
            di.next();
            di.set(si.next());
        }
    }
}
```

8.5 out T 与 in T

正如前面所讲的，在 Java 泛型里有通配符这个东西，我们要用?extends T 指定类型参数的上界，用? super T 指定类型参数的下界。

而 Kotlin 抛弃了这个通配符，直接实现了前面所讲的 PECS 的规则。Kotlin 中引入了投射类型 out T 代表生产者对象，投射类型 in T 代表消费者对象，使用投射类型(projected type)out T 和 in T 来实现与类型通配符同样的功能。

下面通过代码示例简单讲解一下：

```
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
        ...
       ListIterator<? super T> di=dest.listIterator(); //in T，写入 dest 数据
       ListIterator<? extends T> si=src.listIterator();//out T，读取 src 数据
        ...
}
```

List<?super T>dest 是消费数据的对象，数据会被写入 dest 对象中，这些数据对象被“消费吞进肚子里”了（Kotlin 中叫 in T）。

List<?extends T>src 是生产提供数据的对象。src 会“产出”数据（Kotlin 中叫 out T）。

在 Kotlin 中，我们把只能保证读取数据时类型安全的对象叫做生产者，用 out T 标记；把只能保证写入数据安全时类型安全的对象叫做消费者，用 in T 标记。

可以这么记：

out T 等价于 ? extends T；

in T 等价于 ? super T。

8.6 类 型 擦 除

Java 和 Kotlin 的泛型实现，都是采用了运行时类型擦除的方式。也就是说，在运行时，这些类型参数的信息将会被擦除。

泛型是在编译器层次上实现的，生成的 class 字节码文件中是不包含泛型中的类型信息的。例如，在代码中定义的 List<Object>和 List<String>等类型，在编译之后都会变成 List。JVM 看到的只是 List，而由泛型附加的类型信息对 JVM 来说是不可见的。

关于泛型的很多奇怪特性都与这个类型擦除的存在有关，如泛型类并没有自己独有的 Class 类对象。比如 Java 中并不存在 List<String>.class 或是 List<Integer>.class，而只有 List.class。对应地在 Kotlin 中并不存在 MutableList<Fruit>::class，而只有 MutableList:: class。

类型擦除的基本过程也比较简单：

- 首先，找到用来替换类型参数的具体类。这个具体类一般是 Object。如果指定了类型参数的上界的话，则使用这个上界。
- 其次，把代码中的类型参数都替换成具体的类。同时去掉出现的类型声明，即去掉<>的内容。例如，Tget()就变成了 Objectget()，List<String>就变成了 List。
- 最后，根据需要生成一些桥接方法。这是由于擦除了类型之后的类可能缺少某些必须的方法。这个时候就由编译器来动态生成这些方法。

当了解了类型擦除机制之后，我们就会明白是编译器承担了全部的类型检查工作。编译器禁止某些泛型的使用方式，也正是为了确保类型的安全性。

8.7　本 章 小 结

泛型是一个非常有用的东西，尤其在集合类中我们可以发现大量的泛型代码。有了泛型，我们可以拥有更强大、更安全的类型检查，无须手工进行类型转换，并且能够开发更加通用的泛型算法。

第 9 章　文件 I/O 操作、正则表达式与多线程

我们在第 6 章扩展函数与属性中已经介绍过 Kotlin 中类扩展的特性。使用 Kotlin 的扩展函数功能，可以直接为 String 类实现一个 inc()函数，这个函数把字符串中的每一个字符值加 1。

```
"abc".inc() //bcd
```

inc()扩展函数实现如下：

```
fun String.inc(): String {
    var result = ""
    this.map { result += it + 1 }
    return result
}
```

正是因为有了强大的扩展函数，我们可以在 Java 类库的基础上扩展出大量“看似 Java 类中的原生方法”。而实际上 Kotlin 的标准库 kotlin-stdlib 中大量的 API 都是通过扩展 Java 的类来实现的。

本章我们将要介绍的文件 I/O 操作、正则表达式与多线程等相关内容，都是 Kotlin 通过扩展 Java 已有的类来实现的。下面首先来介绍文件的读写操作。

9.1　文件 I/O 操作

Kotlin I/O 操作的 API 在 kotlin.io 包下。Kotlin 的原则就是 Java 已经有的好用的类就直接使用，没有的或者不好用的类，就在原有类的基础上进行功能扩展。例如 Kotlin 就给 File 类写了扩展函数。

Kotlin 为 java.io.File 类扩展了大量好用的扩展函数，这些扩展函数都在 kotlin\io\FileReadWrite.kt 源代码文件中，具体将在后面介绍。

同时，Kotlin 也针对 InputStream、OutputStream 和 Reader 等都做了简单的扩展。它们主要在 kotlin\io\IOStreams.kt、kotlin\io\ReadWrite.kt 源代码文件中。

Koltin 的序列化直接采用了 Java 序列化类的类型别名：

```
internal typealias Serializable = java.io.Serializable
```

下面来简单介绍一下 Kotlin 文件的读写操作。Kotlin 中常用的文件读写 API 如表 9-1 所示。

表 9-1　文件读写 API

函 数 签 名	功 能 说 明
File.readText(charset: Charset = Charsets.UTF_8): String	读取该文件的所有内容作为一个字符串返回
File.readLines(charset: Charset = Charsets.UTF_8): List	读取该文件的每一行内容，存入一个 List 返回
File.readBytes(): ByteArray	读取文件所有内容以 ByteArray 的方式返回
File.writeText(text: String, charset: Charset = Charsets.UTF_8): Unit	覆盖写入 text 字符串到文件中
File.writeBytes(array: ByteArray): Unit	覆盖写入 ByteArray 字节流数组
File.appendText(text: String, charset: Charset = Charsets.UTF_8): Unit	在文件末尾追加写入 text 字符串
File.appendBytes(array: ByteArray): Unit	在文件末尾追加写入 ByteArray 字节流数组

9.1.1　读文件

Kotlin 中提供了使用简单的文件读函数，下面分别介绍。

1．readText:获取文件全部内容字符串

如果简单读取一个文件，可以使用 readText()方法，它直接返回整个文件内容。代码示例如下：

```
fun getFileContent(filename: String): String {
    val f = File(filename)
    return f.readText(Charset.forName("UTF-8")) //获取整个文件的内容，以 UTF-8
                                                 编码格式的字符串
}
```

我们直接使用 File 对象来调用 readText()函数即可获得该文件的全部内容，它返回一个字符串。如果指定字符编码，可以通过传入参数 Charset 来指定，默认是 UTF-8 编码。

2．readLines:获取文件每行的内容

如果想要获得文件中每行的内容，可以简单通过 split("\n")来获得一个每行内容的数组。我们也可以直接调用 Kotlin 封装好的 readLines()函数，获得文件中每行的内容。readLines()函数返回一个持有每行内容的字符串 List。

```
fun getFileLines(filename: String): List<String> {//返回一个持有这个文件中每
                                                   行内容的字符串 List
return File(filename).readLines(Charset.forName("UTF-8"))
}
```

3．readBytes：读取字节流数组

如果希望直接操作文件的字节数组，可以使用 readBytes()函数。

```
//读取为 bytes 数组
val bytes: ByteArray = f.readBytes()  //返回这个文件的字节数组
println(bytes.joinToString(separator = " "))
```

```
//与 Java 互操作，直接调用 Java 中的 InputStream 和 InputStream 类
val reader: Reader = f.reader()
val inputStream: InputStream = f.inputStream()
al bufferedReader: BufferedReader = f.bufferedReader()
```

4. bufferedReader：获取文件的方法签名

获取文件的 bufferedReader()方法签名：

```
fun File.bufferedReader(
    charset: Charset = Charsets.UTF 8,
    bufferSize: Int = DEFAULT BUFFER SIZE
): BufferedReader
```

9.1.2 写文件

使用 Kotlin 扩展的函数，写入文件也变得相当简单。与读取文件类似，我们可以写入字符串，也可以写入字节流，还可以直接调用 Java 的 Writer 或者 OutputStream 类。写文件通常分为覆盖写（一次性写入）和追加写两种情况。

1. writeText：覆盖写文件

我们使用 writeText()函数直接向一个文件中写入字符串 text 的内容：

```
fun writeFile(text: String, destFile: String) {
    val f = File(destFile)
    if (!f.exists()) {
        f.createNewFile()
    }
    f.writeText(text, Charset.defaultCharset()) //覆盖写入字符串 text 的内容
}
```

其中，destFile 参数是目标文件名（带目录）。

2. appendFile：末尾追加写文件

使用 appendFile()函数向一个文件的末尾追加写入内容 text。

```
fun appendFile(text: String, destFile: String) {
    val f = File(destFile)
    if (!f.exists()) {
        f.createNewFile()
    }
    f.appendText(text, Charset.defaultCharset()) //追加写入内容 text
}
```

3. appendBytes：追加写入字节数组

追加字节数组到该文件中方法签名：

```
fun File.appendBytes(array: ByteArray)
```

4．bufferedWriter：获取缓存区写对象

获取该文件的 bufferedWriter()方法签名：

```
fun File.bufferedWriter(
    charset: Charset = Charsets.UTF_8,
    bufferSize: Int = DEFAULT_BUFFER_SIZE
): BufferedWriter
```

提示：关于 Kotlin 对 File 的扩展函数 API 文档，可以参考 https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/java.io.-file/index.html。

9.1.3　遍历文件树

Kotlin 中提供了方便的功能来遍历文件树。

1．walk函数：遍历文件树

下面的例子遍历了指定文件夹下的所有文件。

```
fun traverseFileTree(filename: String) {
    val f = File(filename)
    val fileTreeWalk = f.walk()
    fileTreeWalk.iterator().forEach { println(it.absolutePath) }
                                              //遍历指定文件夹下的所有文件
}
```

测试代码如下：

```
KFileUtil.traverseFileTree(".")
```

上面的测试代码将输出当前目录下的所有子目录及其文件。我们还可以遍历当前文件下所有的子目录文件，将其存入一个 Iterator 中：

```
fun getFileIterator(filename: String): Iterator<File> {
    val f = File(filename)
    val fileTreeWalk = f.walk()
    return fileTreeWalk.iterator()
}
```

我们遍历当前文件下的所有子目录文件，还可以根据条件进行过滤，并把结果存入一个 Sequence 中：

```
fun getFileSequenceBy(filename:String,p:(File)->Boolean):Sequence<File>{
    val f = File(filename)
    return f.walk().filter(p)  //根据条件 p 过滤
}
```

遍历文件树需要调用扩展方法 walk()，它会返回一个 FileTreeWalk 对象，它有一些方法用于设置遍历方向和深度，详情参见 FileTreeWalk API 文档说明。

🔔**提示**：FileTreeWalk API 文档链接地址是 https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/-file-tree-walk/。

2. 递归复制文件

复制该文件或者递归复制该目录及其所有子文件到指定路径下，如果指定路径下的文件不存在，会自动创建。

copyRecursively 函数签名：

```
fun File.copyRecursively(
    target: File,                  //目标文件
    overwrite: Boolean = false, //是否覆盖。true：覆盖之前先删除原来的文件
    onError: (File, IOException) -> OnErrorAction = { _, exception -> throw
     exception }                   //错误处理
): Boolean
```

9.2 网络 I/O

Kotlin 为 Java SDK 中的 java.net.URL 类增加了两个扩展方法，即 readBytes 和 readText（Kotlin 对 Java 已有的 API 做了许多这样的功能扩展与封装）。我们可以方便地使用这两个方法配合正则表达式实现网络爬虫的功能。

下面我们简单写几个函数实例。

根据 URL 获取该 URL 的响应 HTML 函数：

```
fun getUrlContent(url: String): String {
    return URL(url).readText(Charset.defaultCharset())//获取该URL的响应HTML文本
}
```

根据 URL 获取该 URL 响应比特数组函数：

```
fun getUrlBytes(url: String): ByteArray {
    return URL(url).readBytes()          //获取该 URL 的响应 ByteArray
}
```

把 URL 响应字节数组写入文件中：

```
fun writeUrlBytesTo(filename: String, url: String) {
    val bytes = URL(url).readBytes()
    File(filename).writeBytes(bytes)    //写入文件
}
```

下面的例子简单地获取了百度首页的源代码。

```
getUrlContent("https://www.baidu.com")
```

下面的例子根据 URL 来获取一张图片的比特流，然后调用 readBytes()方法读取字节流并写入文件中。

```
writeUrlBytesTo("图片.jpg",
```

```
"http://n.sinaimg.cn/default/4 img/uplaod/3933d981/20170622/2fIE-fyhfxp
h6601959.jpg")
```

在项目相应文件夹下，可以看到下载好的“图片.jpg”。

9.3　执行 Shell 命令

我们使用 Groovy 语言的文件 I/O 操作感觉非常好用，例如：

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4)
class ShellExecuteDemoTest {
    @Test
    def void testShellExecute() {
        def p = "ls -R".execute() //Groovy 中的 shell 执行函数
        def output = p.inputStream.text
        println(output)
        def fname = "我图.url"
        def f = new File(fname)
        def lines = f.readLines()
        lines.forEach({
            println(it)
        })
        println(f.text)
    }
}
```

Kotlin 中的文件 I/O 和网络 I/O 操作与 Groovy 一样简单。另外，从上面的代码中可以看到使用 Groovy 执行终端命令非常简单：

```
def p = "ls -R".execute()
def output = p.inputStream.text
```

在 Kotlin 中，目前还没有对 String 类和 Process 类扩展这样的函数。其实扩展这样的函数非常简单，我们完全可以自己去扩展。

首先我们来扩展 String 的 execute()函数。

```
fun String.execute(): Process {     //给 String 扩展 execute()函数
    val runtime = Runtime.getRuntime()
    return runtime.exec(this)
}
```

然后给 Process 类扩展一个 text()函数。

```
fun Process.text(): String {         //给 Process 类扩展 text()函数
```

```
    var output = ""
    //输出 Shell 执行的结果
    val inputStream = this.inputStream
    val isr = InputStreamReader(inputStream)
    val reader = BufferedReader(isr)
    var line: String? = ""
    while (line != null) {
        line = reader.readLine()
        output += line + "\n"
    }
    return output
}
```

完成了上面两个简单的扩展函数之后，就可以在下面的测试代码中像 Groovy 一样执行终端命令了：

```
val p = "ls -al".execute()

val exitCode = p.waitFor()
val text = p.text()

println(exitCode)
println(text)
```

实际上，通过前面多个实例的学习，我们可以看出 Kotlin 的扩展函数相当实用。Kotlin 语言本身提供的 SDK 标准库 API 中其实也大量使用了扩展功能。

9.4 正则表达式

我们在 Kotlin 中除了仍然可以使用 Java 中的 Pattern、Matcher 等类之外，Kotlin 还提供了一个正则表达式类 kotlin\text\regex\Regex.kt，我们通过 Regex 的构造函数来创建一个正则表达式。

9.4.1 构造 Regex 表达式

使用 Regex 构造函数如下：

```
val r1 = Regex("[a-z]+") //创建一个 Regex 对象，匹配的正则表达式是 [a-z]+
val r2 = Regex("[a-z]+", RegexOption.IGNORE_CASE)
```

其中的匹配选项 RegexOption 是直接使用的 Java 类 Pattern 中的正则匹配选项。

使用 String 的 toRegex()扩展函数如下：

```
val r3 = "[A-Z]+".toRegex() //直接使用 Kotlin 中给 String 扩展的 toRegex 函数
```

9.4.2 Regex 函数

Regex 里面提供了丰富的简单而实用的函数，如表 9-2 所示。

表 9-2　Regex函数

函 数 名 称	功 能 说 明
matches(input: CharSequence): Boolean	输入字符串全部匹配
containsMatchIn(input: CharSequence): Boolean	输入字符串至少有一个匹配
matchEntire(input: CharSequence): MatchResult?	输入字符串全部匹配，返回一个匹配结果对象
replace(input: CharSequence, replacement: String): String	把输入字符串中匹配的部分替换成 replacement 的内容
replace(input: CharSequence, transform: (MatchResult) -> CharSequence): String	把输入字符串中匹配到的值，用函数 transform 映射之后的新值替换
find(input: CharSequence, startIndex: Int = 0): MatchResult?	返回输入字符串中第一个匹配的值
findAll(input: CharSequence, startIndex: Int = 0): Sequence	返回输入字符串中所有匹配的值 MatchResult 的序列

下面分别对表中列举的函数给出简单示例。

1. matches()函数

如果输入的字符串全部匹配正则表达式则返回 true，否则返回 false。

```
>>> val r1 = Regex("[a-z]+")
>>> r1.matches("ABCzxc")    //其中大写的 ABC 不匹配，所以返回 false
false

>>> val r2 = Regex("[a-z]+",RegexOption.IGNORE_CASE)//正则表达式，忽略大小写
>>> r2.matches("ABCzxc")    //忽略大小写，满足正则表达式，所以返回 true
true

>>> val r3 = "[A-Z]+".toRegex()
>>> r3.matches("GGMM")      //都是大写字母，满足[A-Z]正则表达式，所以返回 true
true
```

2. containsMatchIn()函数

如果输入字符串中至少有一个匹配就返回 true，如没有匹配就返回 false。

```
>>> val re = Regex("[0-9]+")
>>> re.containsMatchIn("012Abc")//包含满足条件的就返回 true，012 满足[0-9]+正
                                  则匹配，所以返回 true
true
>>> re.containsMatchIn("Abc")//没有任何字符满足[0-9]+正则表达式，所以返回 false
false
```

3. matchEntire()函数

如果输入字符串全部匹配正则表达式则返回一个 MatcherMatchResult 对象，否则返回 null。

```
>>> val re = Regex("[0-9]+")
>>> re.matchEntire("1234567890") //全部满足匹配条件
kotlin.text.MatcherMatchResult@34d713a2
>>> re.matchEntire("1234567890!")
```

```
null
```

我们可以访问 MatcherMatchResult 的 value 属性来获得匹配的值。

```
>>> re.matchEntire("1234567890")?.value
1234567890
```

由于 matchEntire()函数的返回是 MatchResult? 可空对象，所以这里使用了安全调用符号“?.”。

4. replace(input: CharSequence, replacement: String): String函数

把输入字符串中匹配的部分替换成 replacement 的内容。

```
>>> val re = Regex("[0-9]+")
>>> re.replace("12345XYZ","abcd")
abcdXYZ
```

我们可以看到，12345XYZ 中 12345 是匹配正则表达式[0-9]+的内容，它被替换成了abcd。

5. replace(input: CharSequence, transform: (MatchResult) -> CharSequence):String函数

replace()函数签名如下：

```
replace(input:CharSequence,transform:(MatchResult)->CharSequence):String
```

replace()函数的功能是把输入字符串中匹配到的值，用函数 transform()映射之后的新值进行替换。

```
>>> val re = Regex("[0-9]+")
>>> re.replace("9XYZ8", { (it.value.toInt() * it.value.toInt()).toString() })
81XYZ64
```

可以看到，9XYZ8 中数字 9 和 8 是匹配正则表达式[0-9]+的内容，它们分别被 transform()函数映射(it.value.toInt()*it.value.toInt()).toString()的新值 81 和 64 所替换。

6. find()函数

返回输入字符串中第一个匹配的 MatcherMatchResult 对象。

```
>>> val re = Regex("[0-9]+")
>>> re.find("123XYZ987abcd7777")
kotlin.text.MatcherMatchResult@4d4436d0
>>> re.find("123XYZ987abcd7777")?.value
123
```

7. findAll()函数

返回输入字符串中所有匹配值的 MatchResult 的序列。

```
>>> val re = Regex("[0-9]+")
>>> re.findAll("123XYZ987abcd7777")
kotlin.sequences.GeneratorSequence@f245bdd
```

我们可以通过 forEach 循环遍历所有匹配的值：

```
>>> re.findAll("123XYZ987abcd7777").forEach{println(it.value)}
123
987
7777
```

9.4.3　使用 Java 的正则表达式类

除了前面 Kotlin 提供的函数之外，在 Kotlin 中仍然可以使用 Java 正则表达式的 API。

```
val re = Regex("[0-9]+")
val p = re.toPattern()
val m = p.matcher("888ABC999")
while (m.find()) {
    val d = m.group()
    println(d)
}
```

上面的代码运行后输出如下：

```
888
999
```

9.5　多线程编程

Kotlin 中没有 synchronized、volatile 关键字。Kotlin 的 Any 类似于 Java 的 Object，但是没有 wait()、notify()和 notifyAll()方法。

那么并发如何在 Kotlin 中工作呢？放心，Kotlin 既然是站在 Java 的肩膀上，当然少不了对多线程编程的支持——Kotlin 通过封装 Java 中的线程类，简化了我们的编码。同时我们也可以使用一些特定的注解，直接使用 Java 中的同步关键字等。下面简单介绍一下使用 Kotlin 进行多线程编程的相关内容。

9.5.1　创建线程

我们在 Java 中通常有两种方法创建线程：

扩展 Thread 类或者进行实例化并通过构造函数传递一个 Runnable。因为我们可以很容易地在 Kotlin 中使用 Java 类，所以这两个方式都可以使用。

1．使用对象表达式创建

```
object : Thread() { //object 对象表达式
    override fun run() {
        Thread.sleep(3000)
        println("A 使用 Thread 对象表达式: ${Thread.currentThread()}")
    }
}.start()
```

此代码使用 Kotlin 的对象表达式创建一个匿名类并覆盖 run()方法。

2. 使用Lambda表达式

下面是如何将一个 Runnable 传递给一个新创建的 Thread 实例：

```
Thread({ //Lambda 表达式
    Thread.sleep(2000)
    println("B 使用 Lambda 表达式: ${Thread.currentThread()}")
}).start()
```

我们在这里看不到 Runnable，在 Kotlin 中可以直接使用上面的 Lambda 表达式来表达。还有更简单的方法吗？且看下文解说。

3. 使用Kotlin封装的Thread()函数

例如，我们写了下面一段线程的代码：

```
val t = Thread({
    Thread.sleep(2000)
    println("C 使用 Lambda 表达式:${Thread.currentThread()}")
})
t.isDaemon = false
t.name = "CThread"
t.priority = 3
t.start()
```

后面的四行可以说是样板化的代码。在 Kotlin 中把这样的操作封装简化了。

```
thread(start = true, isDaemon = false, name = "DThread", priority = 3) {
    Thread.sleep(1000)
    println("D 使用 Kotlin 封装的函数 thread(): ${Thread.currentThread()}")
}
```

这样的代码显得更加精简整洁了。事实上，Thread()函数就是对我们编程实践中经常用到的样板化的代码进行了抽象封装，它的实现如下：

```
public fun thread(start: Boolean = true, isDaemon: Boolean = false,
contextClassLoader:
ClassLoader? = null, name: String? = null, priority: Int = -1, block: ()
-> Unit): Thread {
    val thread = object : Thread() {
        public override fun run() {
            block()
        }
    }
    if (isDaemon)
        thread.isDaemon = true
    if (priority > 0)
        thread.priority = priority
    if (name != null)
        thread.name = name
    if (contextClassLoader != null)
        thread.contextClassLoader = contextClassLoader
    if (start)
        thread.start()
    return thread
```

```
}
```

这只是一个使用非常方便的包装函数，简单、实用。从上面的例子中可以看出，Kotlin 通过扩展 Java 的线程 API，简化了样板代码。

9.5.2　同步方法和块

synchronized 不是 Kotlin 中的关键字，它替换为@Synchronized 注解。Kotlin 中的同步方法的声明如下：

```
@Synchronized fun appendFile(text: String, destFile: String) {
    val f = File(destFile)
    if (!f.exists()) {
        f.createNewFile()
    }
    f.appendText(text, Charset.defaultCharset())
}
```

@Synchronized 注解与 Java 中的 synchronized 具有相同的效果，即会将 JVM 方法标记为同步。

对于同步块，我们使用 synchronized()函数，它使用锁作为参数：

```
fun appendFileSync(text: String, destFile: String) {
    val f = File(destFile)
    if (!f.exists()) {
        f.createNewFile()
    }

    synchronized(this){
        f.appendText(text, Charset.defaultCharset())
    }
}
```

synchronized()函数与 Java 的语法基本一样，大家用起来会觉得很熟悉。

9.5.3　可变字段

同样地，Kotlin 中没有 volatile 关键字，但是有@Volatile 注解。

```
@Volatile private var running = false
fun start() {
    running = true
    thread(start = true) {
        while (running) {
            println("Still running: ${Thread.currentThread()}")
        }
    }
}

fun stop() {
    running = false
    println("Stopped: ${Thread.currentThread()}")
}
```

@Volatile 会将 JVM 备份字段标记为 volatile。

当然，在 Kotlin 中还有更好用的协程并发库，在代码工程实践中，可以根据实际情况自由选择。

9.6 本章小结

Kotlin 是一门工程实践性很强的语言，从本章介绍的文件 I/O、正则表达式及多线程等内容中，我们可以领会到 Kotlin 的基本原则：充分使用已有的 Java 生态库，在此基础之上进行更加简单、实用的扩展，大大提升程序员的工作效率。从本章的介绍中我们也体会到了 Kotlin 编程中的极简理念——不断地抽象、封装、扩展，使之更加简单、实用。

本章示例代码地址 https://github.com/EasyKotlin/chapter15_file_io。

另外，笔者综合本章相关的的内容，使用 Kotlin + Spring Boot 写了一个简单的图片+文章的爬虫 Web 应用，感兴趣的读者可参考源码：https://github.com/AK-47-D/cms-spider。

第 10 章　使用 Kotlin 创建 DSL

使用 DSL 的编程风格，可以让程序更加简单、干净、直观。当然，我们也可以创建自己的 DSL。相对于传统的 API，DSL 更加富有表现力，更符合人类的语言习惯。

本章就让我们一起来学习 Kotlin 中 DSL 的相关内容。本章首先会介绍什么是 DSL，然后简单介绍 Kotlin DSL 设计中的特性支持，最后给出一个 HTTP AJAX 请求的 DSL 实现的完整案例。

10.1　什么是 DSL

DSL（Domain-Specific Language，领域特定语言）指的是专注于特定问题领域的计算机语言。不同于通用的计算机语言（GPL），领域特定语言只用在某些特定的领域。

DSL 语言能让我们以一种更优雅、更简洁的方式来表达和解决领域问题。之所以能够这样，是因为 DSL 语言刚好能够用于这个特定的解决领域中存在的模式。

DSL 简单讲就是对一个特定问题（受限的表达能力）的方案模型更高层次的抽象表达（领域语言），使其更加简单易懂（容易理解的语义及清晰的语义模型）。

DSL 只是问题解决方案模型的外部封装，这个模型可能是一个 API 库，也可能是一个完整的框架等。DSL 提供了思考特定领域问题的模型语言，这使得我们可以更加简单、高效地解决问题。DSL 聚焦一个特定的领域，简单易懂，功能极简但完备，更加方便我们理解和使用模型。

比如用来显示网页的 HTML 语言，在 Kotlin 生态中有个 kotlinx.html 可在 Web 应用程序中用于构建 HTML 的 DSL。它可以作为传统模板系统（如 JSP、FreeMarker 等）的替代品。

kotlinx.html 分别提供了 kotlinx-html-jvm 和 kotlinx-html-js 库的 DSL，用于在 JVM 和浏览器（或其他 JavaScript 引擎）中直接使用 Kotlin 代码来构建 HTML，直接解放了原有的 HTML 标签式的前端代码。这样，我们也可以使用 Kotlin 来写传统意义上的 HTML 页面了。Kotlin Web 编程将会更加简单纯净。

提示：更多关于 kotlinx.html 的相关内容，可以参考它的 GitHub 地址 https://github.com/Kotlin/kotlinx.html。

更加典型的例子是用于替代 Android 开发中布局 XML 文件的 DSL 框架 Anko，它使用基于 Kotlin 的 DSL 来声明 Android UI 组件，而不是传统的 XML。

在 Android 中使用下面这样嵌套 DSL 风格的代码来替代 XML 式风格的视图文件：

```
UI {
    //AnkoContext
    verticalLayout {
        padding = dip(30)
        var title = editText {
            //editText 视图
            id = R.id.todo title
            hintResource = R.string.title hint
        }
        var content = editText {
            id = R.id.todo content
            height = 400
            hintResource = R.string.content_hint
        }
        button {
            //button 视图
            id = R.id.todo add
            textResource = R.string.add todo
            textColor = Color.WHITE
            setBackgroundColor(Color.DKGRAY)
            onClick {    -> createTodoFrom(title, content) }
        }
    }
}
```

相比 XML 风格的 DSL（XML 本质上讲也是一种 DSL），使用原生的编程语言（如 Kotlin）DSL 风格更加简单、干净，也更加自由、灵活。

DSL 有内部 DSL 与外部 DSL 之分。例如 Gradle、Anko 等都是使用通用编程语言（Java 和 Kotlin）创建的内部 DSL。

10.1.1 内部 DSL

内部 DSL 是指与项目中使用的通用目的编程语言（Java、C#或 Ruby）紧密相关的一类 DSL。它基于通用编程语言实现。

例如，Rails 框架被称为基于 Ruby 的 DSL，用于管理 Ruby 开发的 Web 应用程序。Rails 之所以被称为 DSL，原因之一在于 Rails 应用了一些 Ruby 语言的特性，使得基于 Rails 编程看上去与基于通用目的的 Ruby 语言编程并不相同。

根据 Martin Fowler 和 Eric Evans 的观点，框架或者程序库的 API 是否满足内部 DSL 的关键特征之一就是它是否有一个流畅（fluent）的接口。这样就能够用短小的对象表达式去组织一个原本很长的表达式，使它读起来更加自然。

10.1.2 外部 DSL

我们已经知道，“内部 DSL”就是利用编程语言自带的语法结构定义出来的 DSL，也叫做“内嵌 DSL”。外部 DSL 是从零开始构建的语言，需要实现语法分析器等。通常情况下，我们只需要实现内嵌式 DSL，因为它更容易构建，并具有很多与外部 DSL 相同的

优势。

外部 DSL 与通用编程语言（GPL）类似，但是外部 DSL 更加专注于特定领域。创建外部 DSL 和创建一种通用的编程语言的过程是相似的，它可以是编译型或者解释型。

提示： 关于 DSL 的详细介绍可以参考《领域特定语言》（Martin Fowler）这本书。

10.2　Kotlin 的 DSL 特性支持

许多现代语言为创建内部 DSL 提供了一些先进的方法，Kotlin 也不例外。在 Kotlin 中创建 DSL，一般主要使用下面 3 个特性：

- 扩展函数、扩展属性；
- 带接收者的 Lambda 表达式（高阶函数）；
- invoke 函数调用约定。

例如上面嵌套 DSL 风格的 UI 示例代码的简单说明，如表 10-1 所示。

表 10-1　嵌套DSL风格的UI代码说明

函　数　名	函 数 签 名	备 注 说 明
UI	fun Fragment.UI(init: AnkoContext.() -> Unit):AnkoContext	android.support.v4.app.Fragment 的扩展函数;入参 init 是一个带接收者的函数字面值，我们直接传入的是一个 Lambda 表达式
verticalLayout	inline fun ViewManager.verticalLayout (init: _LinearLayout.() -> Unit): Linear Layout	android.view.ViewManager 的扩展函数

关于扩展函数和带接收者的函数字面值在前面章节中已经讲过了，不再赘述。这里简单讲一下 Kotlin 中的 invoke 操作符函数。

在前面的集合类章节中，我们知道 Kotlin 中使用下标运算符 foo[x]来等价调用 foo.get(x)操作符函数。同样地，关于 invoke 操作符函数调用有一个类似的约定。我们知道，对一个函数 predicate:(T)->Boolean，可以直接调用 predicate(element)，这样的代码实例可以在 List 的扩展函数 filterTo 中看到：

```
public inline fun <T, C : MutableCollection<in T>> Iterable<T>.filterTo
(destination: C, predicate: (T) -> Boolean): C {
    for (element in this) if (predicate(element)) destination.add(element)
    return destination
}
```

在 Kotlin 中，可以将 foo.invoke()简写成 foo()，在 Kotlin 中操作符是可以重载的，“()”操作符对应的就是类的重载操作符函数 invoke，即此处的 predicate:(T)->Boolean)函数的调用：

```
predicate(element)
```

等价于

```
predicate.invoke(element)
```

上面的是函数类型对象 invoke 函数的例子。而实际上在 Kotlin 中，在类的对象实例中也可以像函数那样直接使用“()”操作符来调用这个类的 invoke 操作符函数。用代码示例来说明可能会更容易理解，下面是一个简单的示例代码：

```
>>> class Hello{
...     operator fun invoke(name:String){
...         println("Hello, $name")
...     }
... }
>>> val hello = Hello()
>>> hello("World") //Hello 对象的 invoke 方法的调用约定写法
Hello, World
>>> hello("Kotlin")
Hello, Kotlin
```

这段代码在 Hello 类中定义了一个操作符函数 invoke，然后我们声明了一个 Hello 类的实例对象 hello，接下来神奇的事情发生了：

```
hello("World")
```

我们直接把这个实例对象 hello 作为函数来调用了：给它传入了参数 World，在 REPL 中运行上面的代码，发现可以正确输出了：

```
>>> hello("World")
Hello, World
```

这个特性一般情况下在程序代码中很少使用到，但是在 DSL 中将会非常有用。这个特性会使得我们的 DSL 代码更加简洁而清晰。

10.3 实现集合类的流式 Kotlin DSL

我们对 Java 的工具类非常熟悉，如 java.util.Collections，这样的类里提供了很多静态方法，例如：

```
public static <T> boolean addAll(Collection<? super T> c, T... elements)
public static <T> int binarySearch(List<? extends Comparable<? super T>>
list, T key)
public static void reverse(List<?> list)
public static void shuffle(List<?> list)
public static <T extends Comparable<? super T>> void sort(List<T> list)
...
```

而在实际编码中，通常这样调用这些静态方法：

```
Collection.reverse(list);
Collection.sort(list);
int index = Collections.binarySearch(list, x);
```

这看起来并不“简单漂亮”。我们期望直接这样调用这些方法：

```
list.sort();
```

```
val index = list.binarySearch(x);
```

这就是 Kotlin 中的扩展函数，我们只要把接收器类型放在其名称的前面：

```
fun <T : Comparable<T>> List<T>.sort() {
  Collections.sort(this)
}
```

这里的 this 参数代表调用该函数的对象（函数接收者）。有了扩展函数，我们就可以开始创建“流式 API”DSL 了。

下面来创建一个给定文件名返回文件中每行文本字符串的流式 API。代码风格是下面这样的：

```
fun main(args: Array<String>) {
    val lines =
        "src/main/resources/data.txt"
            .stream()
            .buffered()
            .reader("utf-8")
            .readLines()

    lines.forEach(::println)
}
```

首先给 String 类型扩展一个 stream()函数：

```
fun String.stream() = FileInputStream(this)
```

然后给 FileInputStream 扩展一个 buffered()函数：

```
fun FileInputStream.buffered() = BufferedInputStream(this)
```

接着给 InputStream 扩展一个 reader(charset: String)函数：

```
fun InputStream.reader(charset: String) = InputStreamReader(this, charset)
```

再给 Reader 扩展一个 readLines()函数：

```
fun Reader.readLines(): List<String> {
    val result = arrayListOf<String>()
    forEachLine {
        result.add(it)
    }
    return result
}
```

有了这些扩展函数，就可以使用上面的流式 API 了。实际上，Kotlin 中的 I/O 文件读写及集合类中的流式 API 就是这么扩展 Java 中的 API 的。

10.4　实现一个 SQL 风格的集合类 DSL

集合类的过滤查询函数有没有可能具备 SQL 一样的风格呢？类似下面这个简单的示例：

```
val queryResult = students.select()
    .where { it.score >= 80 }
```

```
    .and { it.sex == "M" }
```

其中，Student 是一个简单的数据类：

```
data class Student(var name: String, var sex: String, var score: Int)
```

students 变量的初始化值是：

```
val students = listOf(
    Student("jack", "M", 90),
    Student("alice", "F", 70),
    Student("bob", "M", 60),
    Student("bill", "M", 80),
    Student("helen", "F", 100)
)
```

下面我们先来创建 List 的扩展函数 select()，代码如下：

```
fun <E> List<E>.select(): List<E> = this
```

然后实现 where()高阶函数，代码如下：

```
fun <E> List<E>.where(predicate: (E) -> Boolean): List<E> {
    val list = this
    val result = arrayListOf<E>()
    for (e in list) {
        if (predicate(e)) {
            result.add(e)
        }
    }
    return result
}
```

where()函数的实现跟 filter()函数基本一致。

接着实现 and()高阶函数，代码如下：

```
fun <E> List<E>.and(predicate: (E) -> Boolean): List<E> {
    return where(predicate)
}
```

and()函数的过滤条件跟 where()函数的逻辑一致，因此这里直接调用 where(predicate)函数。

完整的代码如下：

```
package com.easy.kotlin.dsl

fun main(args: Array<String>) {

    val students = listOf(
        Student("jack", "M", 90),
        Student("alice", "F", 70),
        Student("bob", "M", 60),
        Student("bill", "M", 80),
        Student("helen", "F", 100)
    )

    val queryResult = students.select()
        .where { it.score >= 80 }
        .and { it.sex == "M" }
```

```
    println(queryResult)
}

fun <E> List<E>.where(predicate: (E) -> Boolean): List<E> {
    val list = this
    val result = arrayListOf<E>()
    for (e in list) {
        if (predicate(e)) {
            result.add(e)
        }
    }
    return result
}

fun <E> List<E>.and(predicate: (E) -> Boolean): List<E> {
    return where(predicate)
}
fun <E> List<E>.select(): List<E> = this
data class Student(var name: String, var sex: String, var score: Int)
```

运行上面的代码，输出如下：

```
[Student(name=jack, sex=M, score=90), Student(name=bill, sex=M, score=80)]
```

10.5　本 章 小 结

相比于 Java，Kotlin 对函数式编程的支持更加友好。Kotlin 的扩展函数和高阶函数（Lambda 表达式），为定义 KotlinDSL 提供了核心的特性支持。

使用 DSL 的代码风格，可以让我们的程序更加直观易懂、简洁优雅。使用 Kotlin 实现一个 DSL 非常简单，而且相当使用。

本章示例代码地址 https://github.com/EasyKotlin/dsl。

另外，在笔者的另一本书《Kotlin 极简教程》的第 14 章中实现了一个类似 jQuery 中 AJAX 的 HTTP 请求的 DSL，感兴趣的读者可以阅读参考。

第 11 章　运算符重载与约定

我们在第 2 章 Kotlin 语法基础中已经学习过关于运算符的相关内容，本章将继续深入探讨 Kotlin 中运算符的重载与约定。

通常一门编程语言中都会内置预定义的运算符（如+、-、*、/、==、!=等），这些运算符的操作对象只能是基本数据类型。而在实际的编程场景中有很多自定义类型，其实也有类似的运算操作。这就是我们通常所说的运算符重载（overload）。

Java 中是不支持运算符重载的。而 Kotlin 支持操作符重载。这些操作符在 Kotlin 中是约定好的固定符号（如加法符号+、乘法符号*）和固定的优先级。而实现这样的操作符，我们也必须使用映射的固定名字的成员函数或扩展函数（加法 plus、乘法 times）。重载操作符的函数需要用 operator 修饰符来标记。

11.1　什么是运算符重载

运算符重载是对已有的运算符赋予新的含义，使同一个运算符作用于不同类型的数据会有对应这个数据类型的行为。

运算符重载的实质是函数重载，本质上是对运算符函数的调用，从运算符到对应函数的映射过程由编译器完成。由于一般数据类型间的运算符没有重载的必要，所以运算符重载主要是面向对象之间的。

Kotlin 中的运算符重载约定定义在 org.jetbrains.kotlin.util.OperatorNameConventions 中：

```
package org.jetbrains.kotlin.util

import org.jetbrains.kotlin.name.Name

object OperatorNameConventions {
    @JvmField val GET_VALUE = Name.identifier("getValue")
    @JvmField val SET_VALUE = Name.identifier("setValue")
    @JvmField val PROVIDE_DELEGATE = Name.identifier("provideDelegate")

    @JvmField val EQUALS = Name.identifier("equals")
    @JvmField val COMPARE_TO = Name.identifier("compareTo")
    @JvmField val CONTAINS = Name.identifier("contains")
    @JvmField val INVOKE = Name.identifier("invoke")
    @JvmField val ITERATOR = Name.identifier("iterator")
    @JvmField val GET = Name.identifier("get")
    @JvmField val SET = Name.identifier("set")
    @JvmField val NEXT = Name.identifier("next")
    @JvmField val HAS_NEXT = Name.identifier("hasNext")
```

```
    @JvmField val COMPONENT REGEX = Regex("component\\d+")

    @JvmField val AND = Name.identifier("and")
    @JvmField val OR = Name.identifier("or")

    @JvmField val INC = Name.identifier("inc")
    @JvmField val DEC = Name.identifier("dec")
    @JvmField val PLUS = Name.identifier("plus")
    @JvmField val MINUS = Name.identifier("minus")
    @JvmField val NOT = Name.identifier("not")

    @JvmField val UNARY_MINUS = Name.identifier("unaryMinus")
    @JvmField val UNARY_PLUS = Name.identifier("unaryPlus")

    @JvmField val TIMES = Name.identifier("times")
    @JvmField val DIV = Name.identifier("div")
    @JvmField val MOD = Name.identifier("mod")
    @JvmField val REM = Name.identifier("rem")
    @JvmField val RANGE_TO = Name.identifier("rangeTo")

    @JvmField val TIMES ASSIGN = Name.identifier("timesAssign")
    @JvmField val DIV ASSIGN = Name.identifier("divAssign")
    @JvmField val MOD ASSIGN = Name.identifier("modAssign")
    @JvmField val REM ASSIGN = Name.identifier("remAssign")
    @JvmField val PLUS ASSIGN = Name.identifier("plusAssign")
    @JvmField val MINUS ASSIGN = Name.identifier("minusAssign")

    @JvmField
    internal val UNARY OPERATION NAMES = setOf(INC, DEC, UNARY PLUS,
     UNARY MINUS, NOT)

    @JvmField
    internal val SIMPLE_UNARY_OPERATION_NAMES = setOf(UNARY_PLUS, UNARY_
     MINUS, NOT)

    @JvmField
    val BINARY_OPERATION_NAMES = setOf(TIMES, PLUS, MINUS, DIV, MOD, REM,
     RANGE_TO)

    @JvmField
    internal val ASSIGNMENT_OPERATIONS = setOf(TIMES_ASSIGN, DIV_ASSIGN,
     MOD_ASSIGN, REM_ASSIGN, PLUS_ASSIGN, MINUS_ASSIGN)
}
```

运算符与操作符函数的映射关系定义在 org.jetbrains.kotlin.types.expressions.OperatorConventions.java 中：

```
public static final ImmutableBiMap<KtSingleValueToken, Name> UNARY
OPERATION NAMES = ImmutableBiMap.<KtSingleValueToken, Name>builder()
            .put(KtTokens.PLUSPLUS, INC)
            .put(KtTokens.MINUSMINUS, DEC)
            .put(KtTokens.PLUS, UNARY PLUS)
            .put(KtTokens.MINUS, UNARY MINUS)
            .put(KtTokens.EXCL, NOT)
            .build();

    public static final ImmutableBiMap<KtSingleValueToken, Name> BINARY_
    OPERATION_NAMES = ImmutableBiMap.<KtSingleValueToken, Name>builder()
            .put(KtTokens.MUL, TIMES)
            .put(KtTokens.PLUS, PLUS)
            .put(KtTokens.MINUS, MINUS)
```

```
            .put(KtTokens.DIV, DIV)
            .put(KtTokens.PERC, REM)
            .put(KtTokens.RANGE, RANGE TO)
            .build();

    public static final ImmutableBiMap<Name, Name> REM TO MOD OPERATION
    NAMES = ImmutableBiMap.<Name, Name>builder()
            .put(REM, MOD)
            .put(REM ASSIGN, MOD ASSIGN)
            .build();

    public static final ImmutableBiMap<KtSingleValueToken, Name> ASSIGNMENT
    OPERATIONS = ImmutableBiMap.<KtSingleValueToken, Name>builder()
            .put(KtTokens.MULTEQ, TIMES ASSIGN)
            .put(KtTokens.DIVEQ, DIV ASSIGN)
            .put(KtTokens.PERCEQ, REM ASSIGN)
            .put(KtTokens.PLUSEQ, PLUS ASSIGN)
            .put(KtTokens.MINUSEQ, MINUS ASSIGN)
            .build();
```

其中，KtTokens.kt 中定义了+、-、*、/、==、!、++、--、*=、/=等运算符的符号。从源码中的这一句：

```
public static final ImmutableSet<KtSingleValueToken> NOT OVERLOADABLE =
        ImmutableSet.of(KtTokens.ANDAND, KtTokens.OROR, KtTokens.ELVIS,
        KtTokens.EQEQEQ, KtTokens.EXCLEQEQEQ);
```

可以知道，Kotlin 中的&&、||、?:、===、!==是不能被重载的。

有了操作符重载，我们可以将两个对象加起来变成另外一个对象。例如，我们自定义一个 BoxInt 类型，然后重载 times（乘法*）函数、plus（加法+）函数。

```
class BoxInt(var i: Int) {
    operator fun times(x: BoxInt) = BoxInt(i * x.i) //使用类成员函数重载

    override fun toString(): String {
        return i.toString()
    }
}

operator fun BoxInt.plus(x: BoxInt) = BoxInt(this.i + x.i)//使用扩展函数的方式重载
```

然后，我们的测试代码如下：

```
fun main(arg: Array<String>) {
    val a = BoxInt(3)
    val b = BoxInt(7)
    println(a + b) //10
    println(a * b) //21
}
```

运算符重载其实是 Kotlin 的一个语法糖。我们可以把上述代码反编译成 Java 字节码，可以看到 a+b 其实是等价于 Java 中的

```
public static final BoxInt plus(@NotNull BoxInt $receiver,@NotNull BoxInt x) {
    ...
    return new BoxInt($receiver.getI() + x.getI());
}
```

下面是 a+b 被编译成 class 代码之后的样子。第 3 行的 INVOKESTATIC 验证了上面的说明：

```
ALOAD 1
ALOAD 2
INVOKESTATIC com/easy/kotlin/OperatorOverloadDemoKt.plus (Lcom/easy/ kotlin/
BoxInt;Lcom/easy/kotlin/BoxInt;)Lcom/easy/kotlin/BoxInt;
POP
```

而 a*b 等价于 Java 中的：

```
public final class BoxInt {
   public final BoxInt times(@NotNull BoxInt x) {
      ...
      return new BoxInt(this.i * x.i);
   }
   ...
}
```

对应的字节码如下。同样地，第 3 行 INVOKEVIRTUAL 表明运算符重载确实是 Kotlin 在编译器层面实现的一个语法糖。

```
ALOAD 1
ALOAD 2
INVOKEVIRTUAL com/easy/kotlin/BoxInt.times (Lcom/easy/kotlin/BoxInt;) Lcom/
easy/kotlin/BoxInt;
POP
```

从上面的例子分析中可以看出，Kotlin 在编译器层面做了大量工作，就是为了让代码更简洁，让编译器处理更多的事情。毋庸置疑的是 Kotlin 的简洁优雅而且强大实用的语法和各种各样的语法糖可以大大地提升程序员的工作效率。这些都是直接使用 Java 享受不到的特性。

11.2　重载二元算术运算符

通过阅读上面的源码，可以总结出 Kotlin 中的二元运算符与重载函数名称之间的映射关系，如表 11-1 所示。

表 11-1　二元运算符与重载函数名称之间的映射关系

二元运算符	重载函数名称	备　注
a + b	a.plus(b)	加法操作
a–b	a.minus(b)	减法操作
a*b	a.times(b)	乘法操作
a/b	a.div(b)	除法操作
a%b	a.rem(b)	取余操作，早期版本中叫 mod
a..b	a.rangeTo(b)	范围操作符

例如，一个简单的 1+1=2 的运算的实例代码如下：

```
>>> 1+1
2
```

其实，本质上执行的是：

```
>>> 1.plus(1)
2
```

Kotlin 中使用 operator fun 声明重载运算符函数。例如上面的 Int 类型的加法运算符函数的声明如下：

```
operator fun plus(other: Byte): Int
```

自定义类型运算符重载函数的作用，与内置赋值运算符的作用是同样的声明方式，但是具体的运算逻辑的实现则是“自定义”的。

编程实例题：设计一个类 Complex，实现复数的基本操作。例如，相加：(1+2i)+(3+4i)=4+6i；相减：(1+2i)−(3+4i)=−2−2i；相乘：(1+2i)*(3+4i)=-5+10i。

我们设计 Complex 类如下。

- ❑ 成员变量：实部 real，虚部 image，均为整数变量；
- ❑ 构造方法：无参构造函数、有参构造函数（参数 2 个）；
- ❑ 成员方法：两个复数的加、减、乘操作。

（1）首先声明一个类 Complex，在里面声明两个 Int 成员变量 real 和 image。代码如下：

```
package com.easy.kotlin

class Complex {
   var real: Int = 0
   var image: Int = 0
}
```

使用 IDEA 进行 Kotlin 编程的过程是非常享受的。直接在当前源码文件 Complex 类内右击，自动生成无参构造函数、2 个参数的构造函数，代码如下：

```
package com.easy.kotlin

class Complex {
   var real: Int = 0
   var image: Int = 0

   constructor()
   constructor(real: Int, image: Int) {
      this.real = real
      this.image = image
   }
}
```

我们看到，Generate 对话框中还可以自动生成 equals()、hashCode()函数和 toString()函数，可以选择 Override Methods、Implement Methods 和自动生成 Copyright 等功能。

（2）实现加法、减法、乘法运算符重载函数。

复数加法的运算规则是：实部加上实部，虚部加上虚部。(a+bi)+(c+di)=(a+c)+(b+d)i 对应的算法实现函数是：

```
operator fun plus(c: Complex): Complex {
   return Complex(this.real + c.real, this.image + c.image)
}
```

复数减法的运算规则是：实部减去实部，虚部减去虚部。(a+bi)−(c+di)=(a−c)+(b−d)i

对应的算法实现函数是：

```
operator fun minus(c: Complex): Complex {
    return Complex(this.real - c.real, this.image - c.image)
}
```

复数乘法的运算规则是按照乘法分配律展开。(a+bi)(c+di)=ac−db+(bc+ad)i 对应的算法实现函数是：

```
 operator fun times(c: Complex): Complex {
     return Complex(this.real * c.real - this.image * c.image, this.real *
     c.image + this.image * c.real)
}
```

为了可读性更好，我们重写 toString()函数如下：

```
override fun toString(): String {
    val img = if (image >= 0) "+ ${image}i" else "${image}i"
    return "$real ${img} "
}
```

测试代码如下：

```
fun main(args: Array<String>) {
    val c1 = Complex(1, 1)
    val c2 = Complex(2, 2)
    val p = c1 + c2
    val m = c1 - c2
    val t = c1 * c2

    println(p)
    println(m)
    println(t)
}
```

输出如下：

```
3 + 3
i-1 -1i
0 + 4i
```

11.3　重载自增自减一元运算符

已经知道 Kotlin 中可以重载的一元运算符如表 11-2 所示。

表 11-2　一元运算符

运算符函数	运　算　符
a.unaryPlus()	+a
a.unaryMinus()	−a
a.not()	!a
a.inc()	a++，++a
a.dec()	a−−，−−a

现在就用实例来说明怎样重载这些运算符。首先定义一个 Point 类，然后实现

unaryMinus 运算符函数的重载。

```
class Point(val x: Int, val y: Int) {
    operator fun unaryMinus() = Point(-x, -y)
    override fun toString(): String {
        return "Point(x=$x, y=$y)"
    }
}
```

测试代码如下：

```
val p1 = Point(1, 1)
println(-p1) //Point(x=-1, y=-1)
```

现在给 Java 中的 BigDecimal 类型添加一个自增运算符 inc()函数，给已有的类添加运算符重载函数，我们采用扩展函数来实现。定义 operator fun BigDecimal.inc()，实现代码如下：

```
operator fun BigDecimal.inc() = this + BigDecimal.ONE
```

然后就可以直接对一个 BigDecimal 类型进行自增的操作了。测试代码如下：

```
    var bigDecimal1 = BigDecimal(100)
var bigDecimal2 = BigDecimal(100)

val tmp1 = bigDecimal1++
val tmp2 = ++bigDecimal2

println(tmp1)           //100
println(tmp2)           //101

println(bigDecimal1)    //101
println(bigDecimal2)    //101
```

因为自增后缀表达式是先返回表达式的值，然后进行加 1 操作，所以 tmp1 的值是 100；而自增前缀表达式是加 1 后值作为表达式的值，所以 tmp2 的值是 101。而在下一行打印变量 bigDecimal1、bigDecimal2 的值都是 101。

类似的自减运算符重载函数实现如下：

```
operator fun BigDecimal.dec() = this - BigDecimal.ONE
```

测试代码如下：

```
var bigDecimal3 = BigDecimal(100)
var bigDecimal4 = BigDecimal(100)
val tmp3 = bigDecimal3—
val tmp4 = --bigDecimal4
println(tmp3)           //100
println(tmp4)           //99
println(bigDecimal3)    //99
println(bigDecimal4)    //99
```

而之所以能在实现函数中直接调用 this+BigDecimal.ONE 和 this-BigDecimal.ONE，是因为 Kotlin 语言本身已经对 BigDecimal 进行了加法、减法、乘法、取余、取负等运算符的重载。这些重载运算符函数定义在 BigNumbers.kt 中。

```
public inline operator fun BigDecimal.plus(other: BigDecimal) : BigDecimal
= this.add(other)

public inline operator fun BigDecimal.minus(other: BigDecimal) : BigDecimal
= this.subtract(other)

public inline operator fun BigDecimal.times(other: BigDecimal) : BigDecimal
= this.multiply(other)

public inline operator fun BigDecimal.div(other: BigDecimal) : BigDecimal
= this.divide(other, RoundingMode.HALF_EVEN)

public inline operator fun BigDecimal.mod(other: BigDecimal) : BigDecimal
= this.remainder(other)

public inline operator fun BigDecimal.rem(other: BigDecimal) : BigDecimal
= this.remainder(other)

public inline operator fun BigDecimal.unaryMinus() : BigDecimal =
this.negate()
```

这些运算符重载函数实现的背后其实就是调用的 BigDecimal 的 add()、subtract()、multiply()、divide()、remainder()、negate()等方法。

可以看出，Kotlin 通过更高层次的封装，大大简化了 BigDecimal 数据类型的算术运算的代码，使得 BigDecimal 算术运算的代码更加简单、易读。而在 Java 中，都不得不使用冗长的方法名进行调用。虽然 Kotlin 背后调用的仍然是 Java 的方法，但是对于 Kotlin 程序员来说，无疑是更加简洁明了了。

11.4　重载比较运算符

我们知道，在 Java 中，“<、>、>=、<=、==、!=”运算符只能作用于基本数据类型的比较。例如：

```
public static void main(String[] args) {
    int x = 1;
    int y = 1;

    boolean b1 = x > y;
    boolean b2 = x < y;
    boolean b3 = x >= y;
    boolean b4 = x <= y;
    boolean b5 = x == y;
    boolean b6 = x != y;

    System.out.println(b1);
    System.out.println(b2);
    System.out.println(b3);
```

```
    System.out.println(b4);
    System.out.println(b5);
    System.out.println(b6);
}
```

而在对象类型上是不允许使用这些比较运算符进行比较的，如图 11-1 所示。

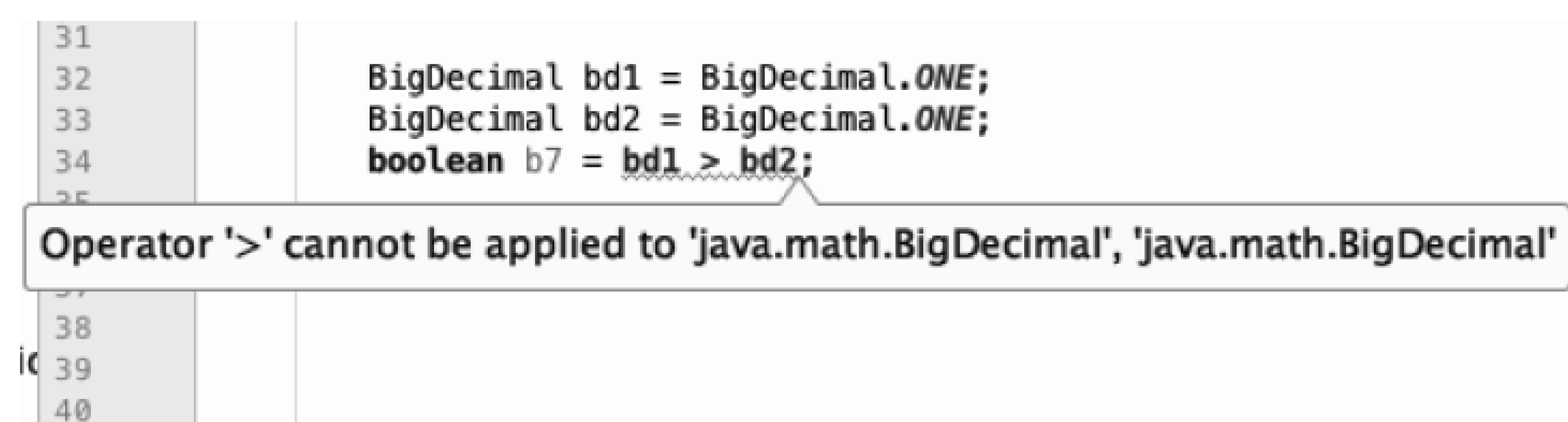


图 11-1　对象类型上不允许使用比较运算符

而实际上，只要给定一个比较标准，原则上对象之间也是可以比较大小的，而不是仅仅限于基本数据类型。因为在 Kotlin 中一切类型都是引用类型。所以，对象之间的比较将是“自然而然”的。本节主要介绍比较运算符的重载。

上面的 BigDecimal 比较的 Java 代码，在 Kotlin 中是允许的：

```
val bd1 = BigDecimal.ONE
val bd2 = BigDecimal.ONE
val bdbd = bd1 > bd2
val bdeq = bd1 == bd2
val bdeqeq = bd1 === bd2
println(bdbd)       //false
println(bdeq)       //true
println(bdeqeq)     //true
```

其中的大于号“>”会映射成调用 compareTo(BigDecimal val)>0 的值。Kotlin 中的比较运算符与重载函数名之间的映射关系如表 11-3 所示。

表 11-3　比较运算符

表　达　式	翻译成函数调用
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0

两个等于号“==”会映射成调用 equals(Object x)方法。需要注意的是，a == b 表达式中就算 a、b 是 null，也可以安全调用。因为 a==b 会被 Kotlin 编译器翻译成带可空性判断的 equals()方法的调用，即 a?.equals(b)?:(b ===null)。

而 3 个等于号“===”是 Kotlin 中自己实现的运算符，这个运算符不能被重载，它不仅会比较值是否相等，还会去比较对象的引用是否相等。因为 BigDecimal.ONE 是常量，在 JVM 内存模型中是存在常量区的，所以 bd1=== bd2 返回的也是 true。

例如，下面的 Point 类：

```
class Point(val x:Int, val y:Int)
```

如果我们不去重载实现其 equals()函数，编译器会自动生成一个 equals()函数与 hashCode()

函数。

```
class Point(val x:Int, val y:Int){
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (javaClass != other?.javaClass) return false

        other as Point
        if (x != other.x) return false
        if (y != other.y) return false

        return true

    }

    override fun hashCode(): Int {
        var result = x
        result = 31 * result + y
        return result
    }
}
```

而如果我们想自定义相等的判断方法，可以进行重写实现其 equals()函数。

现在，我们来比较两个 Point 对象的大小。如果我们定义 Point 对象之间大小的比较标准是其范数（用来度量某个向量空间（或矩阵）中的每个向量的长度）大小，那么比较运算符的重载函数实现如下：

```
operator fun compareTo(other: Point): Int {
    val thisNorm = Math.sqrt((this.x * this.x + this.y * this.y).toDouble())
    val otherNorm = Math.sqrt((other.x * other.x + other.y * other.
    y).toDouble())
    return thisNorm.compareTo(otherNorm)
}
```

测试代码如下：

```
val p1 = Point(1, 1)
val p2 = Point(1, 1)
val p3 = Point(1, 3)
println(p1 >= p2)   //true
println(p3 > p1)    //true
```

11.5　重载计算赋值运算符

计算赋值运算符如表 11-4 所示。

表 11-4　计算赋值运算符

表　达　式	翻译成运算符重载函数的调用
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.remAssign(b)

如果我们想要重载某个类型的这些赋值运算符，只需要实现其对应的运算符重载函数即可。

11.6 本 章 小 结

在进行对象之间的运算时，编译器解析的时候会去调用对应运算符重载函数。为了使代码简单易懂，在实现运算符重载函数的时候一定要考虑其实际问题场景的意义，并且在运算符重载函数上写清楚对象之间的比较规则，注释写清楚。如果滥用运算符重载，会导致代码易读性大大下降。

第 12 章　元编程、注解与反射

反射（Reflection）是在运行时获取类的函数（方法）、属性、父类、接口、注解元数据、泛型信息等类的内部信息的机制。这些信息称之为 RTTI（Run-Time Type Information，运行时类型信息)。

注解（Annotation）是我们给代码添加的元数据。使用注解可以写出更加简洁、干净的代码，同时还可以在编译期进行类型检查。Kotlin 的注解完全兼容 Java 的注解。

本章主要介绍 Kotlin 中注解与反射编程的相关内容。

12.1　元编程简介

说到元编程（Meta-programming），要先从 Meta-这个前缀开始说起。Meta-前缀在西方哲学界是指：关于事物自身的事物。例如，心理学领域有一门专门研究关于人类认知心理的学科叫认知心理学（cognitive psychology），还有一门学科是研究人类对自己的认知过程的认知，叫做元认知心理学（Meta-cognitive psychology ），又称反省认知、监控认知、超认知、反审认知等。元认知的本质是人类对自身认知活动的自我意识和自我调节。

再例如，Meta-knowledge 就是“关于知识本身的知识”，Meta-data 就是“关于数据的数据”，Meta-language 就是“关于语言的语言”，而 Meta-programming 就是“关于编程的编程”，也就是我们通常所说的“元编程”。

元编程（Meta-programming）是指用代码在编译期或运行期生成或改变代码的一种编程形式。编写元程序的语言称之为元语言，被操纵的语言称之为目标语言。如果一门语言中具备同时是元语言也是目标语言的能力，这就是反射。

一般代码的操作对象是数据，元编程操作的对象是其他代码，无关业务逻辑，只跟当前代码结构相关的代码。例如，在 Java 中运行时通过反射把所有以*ServiceImpl 结尾的类找出来，加上 log 日志或者进行监控统计等其他动作。除非程序运行期的输入数据会被直接或间接转化成代码，否则元编程不会给程序带来新的逻辑。

元编程本质上是一种对源代码本身进行高层次抽象的编码技术。元编程比我们手写的代码多提供了一个抽象层次！我们其实就是用代码中的元数据（按照一定的协议规则来定义，也就是注解的语法规范）来动态插入新代码逻辑，就是用来动态生成代码的程序。其实，根本没有什么“元编程”，有的只是“编程”。

反射是促进元编程的一种很有价值的语言特性。编程语言中的泛型支持也使用元编程能力。元编程通常有两种方式：一种是通过应用程序接口（API）来暴露运行时系统的内

部信息；另一种方法是在运行时动态执行包含编程命令的字符串。因此，“程序能编写程序”。虽然两种方法都能用，但大多数时候主要用其中一种。

注解是把编程中的元数据信息直接写在源代码中，而不是保存在外部文件中。

在使用注解之前（甚至在使用之后），XML 配置文件被广泛应用于编程过程中对元数据的描述。后来程序员们逐渐发现 XML 的维护越来越糟糕了，进而希望直接使用一些和代码紧耦合的“元数据”，而不是像 XML 那样和代码分离。在把注解使用得淋漓尽致的 Spring Boot 框架中，基本不需要一行 XML 配置，几乎全部使用注解就可以完成一个 Spring 企业级应用的开发。

XMLvs.Annotation，这其实是一个“阴阳交融”的编程之道，很多时候要看具体的问题场景来决定采用哪种方式。XML 配置就是为了分离代码和配置而引入的，而注解是为了希望使用一些和代码紧耦合的东西。事物的发展就是这样阴阳交合、辩证发展的过程。

注解是将元数据附加到代码的方法。而反射可以在运行时把代码中的注解元数据获取到，并在目标代码执行之前进行动态代理，实现业务逻辑的动态注入，这其实就是 AOP（Aspect Oriented Programming，面向切面编程（也叫面向方面））的核心思想——通过运行期动态代理（和预编译方式）实现在不修改源代码的情况下，给程序动态添加新功能的一种技术。

例如，在 Spring、Mybatis、JPA 等诸多框架中的核心功能都是使用了注解与反射的技术来实现的。例如常用的 Spring 框架中的各种注解 @Repository、@Service、@Transactional、@RequestMapping、@ResponseBody 等，以及 Mybatis 框架中的各种注解@Select、@Update、@Param 等。

另外，需要重点提到的就是当下非常流行的 Spring Boot 框架。在使用 Spring Boot 框架开发企业级应用时，完全不需要使用一行 XML 配置，整个源代码工程都能基于注解来开发（application.propertis 配置文件另当别论），更多关于 SpringBoot 框架开发的知识，将在后面的章节中介绍。

12.2 注　　解

Kotlin 的注解跟 Java 注解也完全兼容。我们可以在 Kotlin 代码中很自然地使用 Java 中的注解。也就是说，我们使用 Kotlin 语言集成 SpringBoot 框架开发的过程将会非常自然，几乎与使用原生 Java 语言开发一样流畅，同时还能享受 Kotlin 语言带来的诸多简洁且非常强大的特性。

12.2.1 声明注解

Kotlin 中声明注解使用 annotation class 关键字。例如，我们声明两个注解 Run 和 TestCase 如下：

```
@Target(AnnotationTarget.CLASS,
```

```
        AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE PARAMETER,
        AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.RUNTIME)
@Repeatable
@MustBeDocumented
annotation class TestCase(val id: String)


@Target(AnnotationTarget.CLASS,
        AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER,
        AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.RUNTIME)
@Repeatable
@MustBeDocumented
annotation class Run
```

从这个关键字上可以看出注解也是一种 class，编译器同样可以对注解类型在编译期进行类型检查。

自定义的注解中使用的注解（例如@Target、@Retention 等），称之为元注解（Meta-annotation）。通过向注解类添加元注解的方法来指定其他属性。元注解说明如表 12-1 所示。

表 12-1　元注解说明

元注解名称	功 能 说 明
@Target	指定这个注解可被用于哪些元素（这些元素定义在 kotlin. annotation. AnnotationTarget 枚举类中。它们是：类 CLASS、注解类 ANNOTATION_CLASS、泛型参数 TYPE_PARAMETER、函数 FUNCTION、属性 PROPERTY，用于描述域成员变量的 FIELD、局部变量 LOCAL_VARIABLE、VALUE_PARAMETER、CONSTRUCTOR、PROPERTY_GETTER、PROPERTY_SETTER，以及用于描述类、接口（包括注解类型）或 enum 声明的 TYPE、表达式 EXPRESSION、文件 FILE、类型别名 TYPEALIAS 等
@Retention	指定这个注解的信息是否被保存到编译后的 class 文件中，以及在运行时是否可以通过反射访问到它。可取的枚举值有 3 个，分别是：SOURCE（注解数据不存储在二进制输出）、BINARY（注解数据存储在二进制输出中，但反射不可见）、RUNTIME（注解数据存储在二进制输出中，可用于反射（默认值）
@Repeatable	允许在单个元素上多次使用同一个注解
@MustBeDocumented	表示这个注解是公开 API 的一部分，在自动产生的 API 文档的类或者函数签名中，应该包含这个注解的信息

12.2.2　使用注解

上面我们声明了 Run 注解，它可以使用在 CLASS、FUNCTION、VALUE_PARAMETER 和 EXPRESSION 上。我们这里给出的示例是用在类上：

```
@Run
class SwordTest {}
```

我们声明的 TestCase 注解有个构造函数，传入的参数是一个 String 类型的 ID。把这个注解用在函数上：

```
@Run
class SwordTest {
    @TestCase(id = "1")
    fun testCase(testId: String) {
        println("Run SwordTest ID = ${testId}")
    }
}
```

上面是注解在代码中的简单使用示例。其中的@TestCase(id="1")是注解构造函数的使用。注解可以有带参数的构造器。注解参数可支持的数据类型如下：

- 基本数据类型（Int,Float,Boolean,Byte,Double,Char,Long,Short)；
- String 类型；
- KClass 类型；
- enum 类型；
- Annotation 类型。

以上为所有引用类型的数组（注意，不包括基本数据类型）。

例如下面都是合法的注解构造函数的参数类型：

```
annotation class TestCase(val id: String)
annotation class TestCasee(val id: Int)
annotation class TestCaseee(val id: Array<String>)
annotation class TestCaseeee(val id: Run)
annotation class TestCaseeeeee(val id: KClass<String>)
```

而下面的两种声明编译不通过：

```
annotation class TestCaseeeee(val id: Array<Int>)
annotation class TestCaseeeeee(val id: SwordTest)
```

另外需要注意的是，注解类型中不能有 null 类型，因为 JVM 不支持将 null 作为注解属性的值进行存储。如果注解用作另一个注解的参数时，则其名称不能以@字符为前缀。例如：

```
annotation class AnnoX(val value: String)

annotation class AnnoY(
       val message: String,
       val annoX: AnnoX = AnnoX("X"))
```

Java 注解与 Kotlin 完全兼容。下面是一个 Kotlin 使用 JUnit4 进行单元测试代码编写的例子。

```
package com.easy.kotlin

import com.easy.kotlin.annotation.SwordTest
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4::class)
class AnnotationClassNoteTest {
    @Test
    fun testAnno() {
        val sword = SwordTest()
```

```
        sword.testCase("10000")
    }
}
```

可以看出，除了@RunWith(JUnit4::class)的反射写法稍微有点不同外，剩下的与我们在 Java 中使用 JUnit 的注解方式基本上是一样的。

12.2.3　处理注解

定义了注解，并在需要的时候给相关类和类属性加上注解信息后，如果没有相应的注解信息处理逻辑流程，那么注解可以说是废了，没有什么实用价值。如何让注解在程序运行的时候发挥其特有的作用呢？核心就在于注解处理的代码了。本节我们将学习怎样进行注解信息的获取和处理。因为注解信息的获取主要是使用反射 API，所以也会在本节中讲到反射相关的内容。

首先，我们的目标测试类是：

```
@Run
class SwordTest {

    @TestCase(id = "1")
    fun testCase(testId: String) {
        println("Run SwordTest ID = ${testId}")
    }
}
```

这里我们主要介绍@TestCase 注解作用在函数上的处理过程。

1．::class引用

首先声明一个变量指向 SwordTest 对象实例：

```
val sword = SwordTest()
```

然后就可以通过这个变量来获取该对象的类的信息。使用::class 来获取 sword 对象实例的 KClass 类的引用。

```
val kClass = sword::class
```

上面的这行代码，Kotlin 编译器会自动推断出 kClass 变量的类型是

```
val kClass:KClass<out SwordTest> = sword::class
```

这个 KClass 数据类型将在后面的小节中介绍。

2．declaredFunctions 扩展属性

下面我们需要获取 sword 对象类型所声明的所有函数。Kotlin 中可以直接使用扩展属性 declaredFunctions 来获取这个类中声明的所有函数（对应的反射数据类型是 KFunction）。代码如下：

```
val declaredFunctions = kClass.declaredFunctions
```

返回的是一个 Collection>，其中，<*>是 Kotlin 泛型中的星投影，类似 Java 中的<?>通配符。

declaredFunctions 扩展属性的实现源码如下：

```
@SinceKotlin("1.1")
val KClass<*>.declaredFunctions: Collection<KFunction<*>>
    get() = (this as KClassImpl).data().declaredMembers.filterIsInstance
     <KFunction<*>>()
```

3. annotations属性

KFunction 类型继承了 KCallable，KCallable 又继承了 KAnnotatedElement。KAnnotated Element 中有个 public val annotations:List 属性里存储了该函数所有的注解信息。通过遍历这个存储 Annotation 的 List，可以获取到 TestCase 注解：

```
for (f in declaredFunctions) {
        //处理 TestCase 注解，使用其中的元数据
        f.annotations.forEach {
            if (it is TestCase) {
                val id = it.id    //TestCase 注解的属性 ID
                doSomething(id)   //注解处理逻辑
            }
        }
}
```

4. call函数

另外，如果想通过反射来调用函数，可以直接使用 call()函数：

```
f.call(sword, id)
```

上面的代码等价于：

```
f.javaMethod?.invoke(sword, id)
```

到这里，我们就完成了一个简单的注解处理器。完整的代码如下：

```
fun testAnnoProcessing() {
    val sword = SwordTest()
    // val kClasss:KClass<out SwordTest> = sword::class //类型声明可省略
    val kClass = sword::class

    val declaredFunctions = kClass.declaredFunctions //获取 sword 对象类型所
                                                       声明的所有函数
    println(declaredFunctions)

    for (f in declaredFunctions) {
        //处理 TestCase 注解，使用其中的元数据
        f.annotations.forEach {
            if (it is TestCase) {
                val id = it.id
                doSomething(id)    //注解处理逻辑
                f.call(sword, id) //等价于 f.javaMethod?.invoke(sword, id)
            }
```

```
        }
    }
}

private fun doSomething(id: String) {
    println("Do Something in Annotation Processing ${id} ${Date()} ")
}

@Target(AnnotationTarget.CLASS,
        AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER,
        AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.RUNTIME)
@Repeatable
@MustBeDocumented
annotation class TestCase(val id: String)

class SwordTest {

    @TestCase(id = "1")
    fun testCase(testId: String) {
        println("Run SwordTest ID = ${testId}")
    }
}
```

测试代码如下：

```
fun main(args: Array<String>) {
    testAnnoProcessing()
}
```

输出如下：

```
[fun com.easy.kotlin.annotation.SwordTest.testCase(kotlin.String): kotlin.Unit]
Do Something in Annotation Processing 1 Mon Oct 23 23:04:09 CST 2017
Run SwordTest ID = 1
```

12.3　反　　射

在前面的注解信息的获取与处理逻辑的实现中，其实已经用到了反射。反射是指在运行时（Run Time），程序可以访问、检测和修改它本身状态或行为的一种能力。Kotlin 中的函数和属性也是头等公民，我们可以通过反射来内省属性和函数，如运行时属性名或类型，函数名或类型等。

在 Kotlin 中我们有两种方式来实现反射的功能。一种是调用 Java 的反射包 java.lang.reflect 下面的 API，另外一种方式就是直接调用 Kotlin 语言提供的 kotlin.reflect 包下面的 API。不过因为反射功能的应用场景并非所有编程场景都用到，所以 Kotlin 把 kotlin.reflect 包的实现放到了单独的 kotlin-reflect-1.1.50.jar（当前版本号是 1.1.50）里面。在实际工程中，如果需要使用 Kotlin 的反射功能，以 Gradle 为例，需要在 build.gradle 配置文件中添加以下依赖：

```
compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"
```

Kotlin 反射 API 类的层次结构如图 12-1 所示。

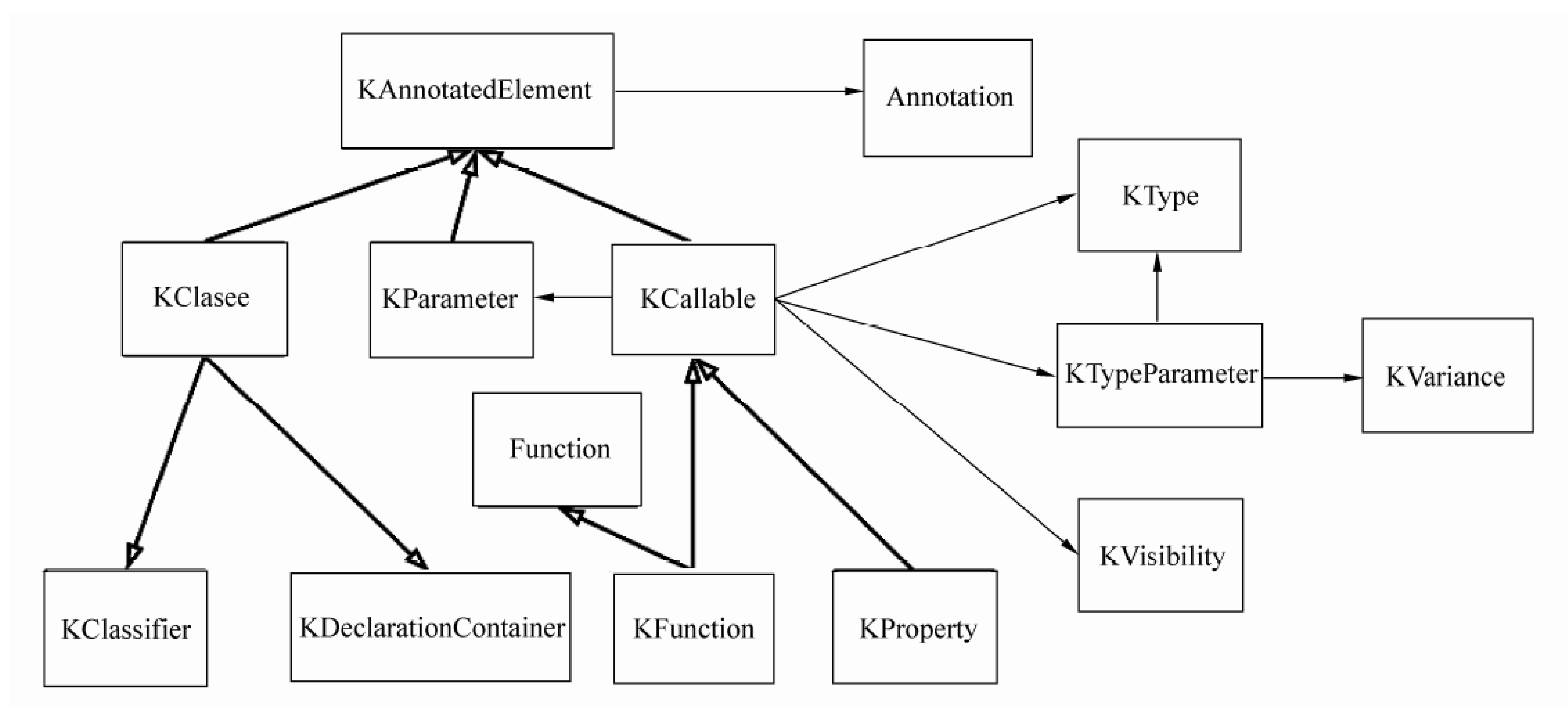


图 12-1　Kotlin 反射 API 类的层次结构

12.3.1　类引用

为了方便讲解，首先定义一个代码实例：

```
open class BaseContainer<T>

class Container<T : Comparable<T>> : BaseContainer<Int> {
    var elements: MutableList<T>

    constructor(elements: MutableList<T>) {
        this.elements = elements
    }

    fun sort(): Container<T> {
        elements.sort()
        return this
    }

    override fun toString(): String {
        return "Container(elements=$elements)"
    }
}
```

反射是在运行时获取一个类引用。我们已经知道使用::class 调用可以获取到当前对象的 KClass 对象：

```
val container = Container(mutableListOf<Int>(1, 3, 2, 5, 4, 7, 6))
val kClass = container::class //获取 KClass 对象
```

需要注意的是，Kotlin 中类引用和 Java 中的类引用是不同的，要获得 Java 类的引用，

可以直接使用 javaClass 这个扩展属性。

```
val jClass = container.javaClass //获取 Java Class 对象
```

javaClass 扩展属性在 Kotlin 中的实现源码如下：

```
public inline val <T: Any> T.javaClass : Class<T>
    @Suppress("UsePropertyAccessSyntax")
    get() = (this as java.lang.Object).getClass() as Class<T>
```

或者使用 KClass 实例的.java 属性：

```
val jkCLass  = kClass.java
```

KClass.java 的扩展属性实现源码如下：

```
@Suppress("UPPER_BOUND_VIOLATED")
public val <T> KClass<T>.java: Class<T>
    @JvmName("getJavaClass")
    get() = (this as ClassBasedDeclarationContainer).jClass as Class<T>
```

12.3.2　函数引用

例如，有一个简单地判断一个 Int 整数是否是奇数的函数：

```
fun isOdd(x: Int) = x % 2 != 0
```

可以在代码中直接调用：

```
>>> isOdd(7)
true
>>> isOdd(2)
false
```

另外，在高阶函数中如想把它当作一个参数来使用，可以使用“::”操作符。例如：

```
val nums = listOf(1, 2, 3)
val filteredNums = nums.filter(::isOdd)
println(filteredNums) //[1, 3]
```

这里的::isOdd 就是一个函数类型(Int)->Boolean 的值。

12.3.3　属性引用

在 Kotlin 中，访问属性属于第一级对象，可以使用“::”操作符：

```
var one = 1
fun testReflectProperty() {
    println(::one.get())    //1
    ::one.set(2)
    println(one)            //2
}

fun main(args: Array<String>) {
    testReflectProperty()
}
```

表达式::one 等价于类型为 KProperty 的一个属性，它可以允许我们通过 get()函数获取值::one.get()。

对于可变属性 varone=1，返回类型为 KMutableProperty 的值，并且还有 set 方法::one.set(2)。

12.3.4 绑定函数和属性引用

我们可以引用一个对象实例的方法。例如下面的代码：

```
val digitRegex = "\\d+".toRegex()
digitRegex.matches("7") //true
digitRegex.matches("6") //true
digitRegex.matches("5") //true
digitRegex.matches("X") //false
```

其中，digitRegex.matches 重复出现，显得比较“样板化”。在 Kotlin 中可以直接引用 digitRegex 对象实例的 matches()方法。上面的代码可以写成下面这样：

```
val isDigit = digitRegex::matches   //引用 digitRegex 对象实例的 matches()方法
isDigit("7")                        //true
isDigit("6")                        //true
isDigit("5")                        //true
isDigit("X")                        //true
```

是不是很酷？真的是相当简洁。

12.4 使用反射获取泛型信息

在 Java 中，使用反射的一个代码实例如下：

```
package com.easy.kotlin;

import java.lang.annotation.Annotation;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Arrays;
import java.util.List;

interface StudentService<T> {
    List<T> findStudents(String name, Integer age);
}

public class ReflectionDemo {
    public static void main(String[] args) {
        StudentServiceImpl studentService = new StudentServiceImpl();
        studentService.save(new Student("Bob", 20));
        studentService.findStudents("Jack", 20);

       //反射 API 调用示例
       final Class<? extends StudentServiceImpl> studentServiceClass =
        studentService.getClass();
```

```
        Class<?>[] classes = studentServiceClass.getDeclaredClasses();
        Annotation[] annotations = studentServiceClass.getAnnotations();
        ClassLoader classLoader = studentServiceClass.getClassLoader();
                                //Returns the class loader for the class
        Field[] fields = studentServiceClass.getDeclaredFields();
                                                       //获取类成员变量
        Method[] methods = studentServiceClass.getDeclaredMethods();
                                                       //获取类成员方法
        try {
            methods[0].getName(); //save
            methods[0].invoke(studentService, "Jack",20);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}

class   StudentServiceImpl   extends   BaseService<Student>   implements
StudentService<Student> {
    public List<Student> findStudents(String name, Integer age) {
        return Arrays.asList(new Student[] {new Student("Jack", 20), new
         Student("Rose", 20)});
    }

    @Override
    public int save(Student student) {
        return 0;
    }
}

abstract class BaseService<T> {
    abstract int save(T t);
}

class Student {

    String name;
    Integer age;

    public Student(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
```

```
}
```

通过反射，可以获取一个类的注解、方法、成员变量等。那么能不能通过反射获取到泛型的信息呢？我们知道 Java 中的泛型采用擦拭法。在程序运行时，无法得到自己本身的泛型信息。而当这个类继承了一个父类，父类中有泛型的信息时，那么就可以通过调用 getGenericSuperclass()方法得到父类的泛型信息。getGenericSuperclass()是 Generic 继承的特例，对于这种情况子类会保存父类的 Generic 参数类型，返回一个 ParameterizedType。另外，我们所说的 Java 泛型在字节码中会被擦除，并不总是擦除为 Object 类型，而是擦除到上限类型。

在 Kotlin 也是一样的泛型机制。所以，通过反射能拿到的也只能是有继承父类泛型信息的子类泛型。具体的代码示例如下：

```
class A<T>

open class C<T>
class B<T> : C<Int>()  //继承父类 C<Int>()

fun fooA() {
    //无法在此处获得运行时 T 的具体类型！下面的代码运行时会报错
    val parameterizedType = A<Int>()::class.java.genericSuperclass as
    ParameterizedType
    val actualTypeArguments = parameterizedType.actualTypeArguments
    for (type in actualTypeArguments) {
        val typeName = type.typeName
        println("typeName=${typeName}") //运行会报错：java.lang.Class cannot
                                be cast to java.lang.reflect.ParameterizedType

    }
}

fun fooB() {
    //当继承了父类 C<Int> 的时候，在此处能够获得运行时 genericSuperclass T 的具体类型
    val parameterizedType = B<Int>()::class.java.genericSuperclass as
    ParameterizedType
    val actualTypeArguments = parameterizedType.actualTypeArguments
    for (type in actualTypeArguments) {
        val typeName = type.typeName
        println("typeName = ${typeName}") //输出：typeName = java.lang.Integer
    }
}

fun main(args: Array<String>) {
    fooA()
    fooB()
}
```

下面通过一个简单的实例来说明 Kotlin 中的反射怎样获取泛型代码的基本信息。

首先声明一个父类 BaseContainer：

```
open class BaseContainer<T>
```

然后声明一个 Container> 继承该父类：

```
class Container<T : Comparable<T>> : BaseContainer<Int> {
    var elements: MutableList<T>

    constructor(elements: MutableList<T>) {
        this.elements = elements
    }

    fun sort(): Container<T> {
        elements.sort()
        return this
    }

    override fun toString(): String {
        return "Container(elements=$elements)"
    }
}
```

再声明一个 Container 对象实例：

```
val container = Container(mutableListOf<Int>(1, 3, 2, 5, 4, 7, 6))
```

然后获取 container 的 KClass 对象引用：

```
val kClass = container::class //获取 KClass 对象
```

KClass 对象的 typeParameters 属性中存有类型参数的信息，代码示例如下：

```
val typeParameters = kClass.typeParameters //获取类型参数 typeParameters 信息，也即泛型信息
```

这个 typeParameters 是一个数组，我们取第一个对象：

```
val kTypeParameter: KTypeParameter = typeParameters[0]
```

对象 kTypeParameter 的属性有 name、isReified、upperBounds 和 variance 等，代码示例如下：

```
println(kTypeParameter.isReified)       //输出：false
println(kTypeParameter.name)            //输出：T
println(kTypeParameter.upperBounds)     //输出：[kotlin.Comparable<T>]
println(kTypeParameter.variance)        //输出：INVARIANT
```

KClass 的 constructors 属性中存有构造函数的信息，可以从中获取构造函数的入参等信息。

```
val constructors = kClass.constructors
for (KFunction in constructors) {
    KFunction.parameters.forEach {
        val name = it.name
        val type = it.type
        println("name = ${name}") //输出：elements
        println("type = ${type}") //输出：kotlin.collections.MutableList<T>
        for (KTypeProjection in type.arguments) {
            println(KTypeProjection.type) //输出：T
```

```
        }
    }
}
```

12.5 本 章 小 结

使用注解与反射可以编写出功能强大的代码，可以在运行时动态分析获取类的结构信息，可以对类进行自查。Kotlin 的反射 API 与 Java 基本类似，在一些细节上有点差异。在前面的章节中已经学习了 Kotlin 语言本身的知识，在后面的章节中将介绍使用 Kotlin+Spring Boot 进行服务端开发、 使用 Kotlin 进行 Android 移动端开发等相关内容。

第 13 章　Kotlin 集成 Spring Boot 服务端开发

本章介绍 Kotlin 服务端开发的相关内容。首先简单介绍一下 Spring Boot 服务端开发框架，快速给出一个 Restful Hello World 的示例。然后讲下 Kotlin 集成 Spring Boot 进行服务端开发的步骤，最后给出一个完整的 Web 应用开发实例。

13.1　用 Spring Boot 快速开发 Restful Hello World

Spring Boot 大大简化了使用 Spring 框架过程中各种烦琐的配置。另外可以更加方便地整合常用的工具链（如 Redis、Email、kafka、ElasticSearch、MyBatis 和 JPA）等，缺点是集成度较高（事物都是两面性的），使用过程中不容易了解底层，遇到问题时解决曲线比较陡峭。本节将介绍怎样快速开始 Spring Boot 服务端的开发。

13.1.1　Spring Initializr

工欲善其事必先利其器。我们使用 https://start.spring.io/可以直接自动生成 Spring Boot 项目脚手架，如图 13-1 所示。

图 13-1　使用 Spring Initializr 生成项目

单击 Switch to the full version 链接，可以看到脚手架支持的工具链。我们也可以自己搭建本地的 Spring Initializr 服务，步骤如下：

（1）Gitclone 源码到本机 https://github.com/spring-io/initializr。

（2）在源码根目录下执行$./mvnw clean install。

（3）到 initializr-service 子项目的目录下

```
cd initializr-service
```

执行

```
../mvnw spring-boot:run
```

即可看到启动日志：

```
......
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080(http)
i.s.i.service.InitializrService          : Started InitializrService in
15.192 seconds (JVM running for 15.882)
```

此时，用本机浏览器访问 http://127.0.0.1:8080/，即可看到脚手架 initializr 页面。

13.1.2　创建 Spring Boot 项目

下面使用本地搭建的脚手架 initializr 来创建基于 Gradle 构建的 Kotlin + Spring Boot 项目，如图 13-2 所示。

图 13-2　创建基于 Gradle 构建的 Kotlin + Spring Boot 项目

首先，我们选择生成的是一个使用 Gradle 构建的 Kotlin 项目，SpringBoot 的版本号这里选择 2.0.0(SNAPSHOT)。

在 Spring Boot Starters 和 dependencies 选项中，选择 Web starter，这个启动器里面包含了基本够用的 Spring Web 开发需要的东西：Tomcat 和 Spring MVC。

其余的项目元数据（Project Metadata）的配置（Bill Of Materials），可以从图 13-2 中看到。然后单击 Generate Project 按钮，会自动下载一个项目的 zip 压缩包，将其解压导入 IDEA 中，如图 13-3 所示。

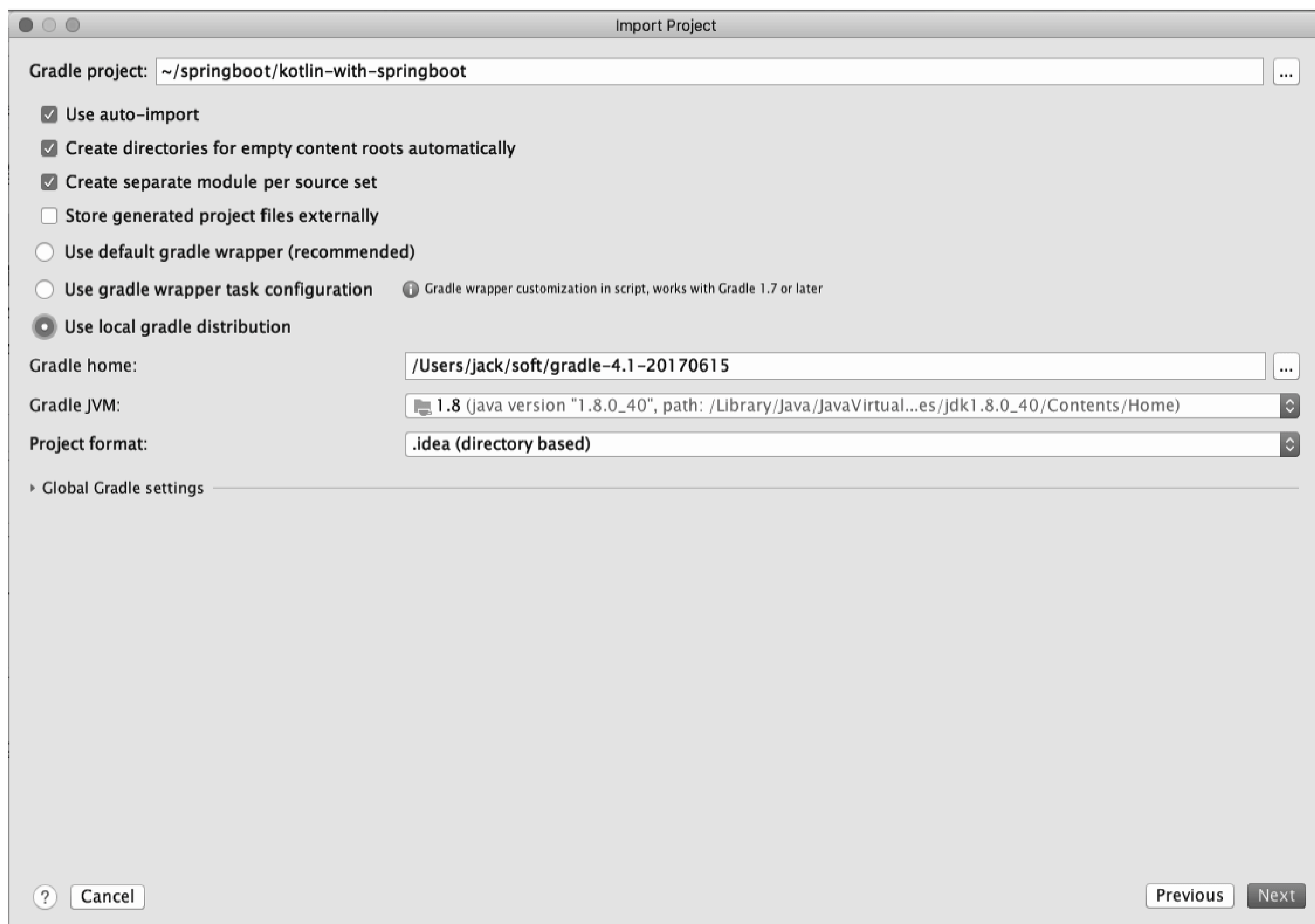


图 13-3　解压工程导入 IDEA 中

因为我们使用的是 Gradle 构建项目，所以需要配置一下 Gradle 环境。这里使用的是 Local gradle distribution，因此选择对应本地的 Gradle 软件包目录。

1．工程文件目录树

我们将得到下面的一个样板工程，工程文件目录树如下：

```
kotlin-with-springboot$ tree
.
├── build
│   └── kotlin-build
│       └── version.txt
├── build.gradle
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
```

```
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── kotlin-with-springboot.iml
└── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               └── kotlinwithspringboot
    │   │                   └── KotlinWithSpringbootApplication.kt
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        ├── java
        ├── kotlin
        │   └── com
        │       └── easy
        │           └── kotlin
        │               └── kotlinwithspringboot
        │                   └── KotlinWithSpringbootApplicationTests.kt
        └── resources

23 directories, 10 files
```

其中，src\main\kotlin 是 Kotlin 源码放置目录。src\main\resources 目录下面放置工程资源文件。application.properties 是工程全局的配置文件，static 文件夹下面放置的是静态资源文件，templates 目录下面放置的是视图模板文件。

2. build.gradle 配置文件

我们使用 Gradle 来构建项目。其中，build.gradle 配置文件类似 Maven 中的 pom.xml 配置文件。使用 Spring Initializr 自动生成的样板项目的默认配置如下：

```
buildscript {
    ext {
        kotlinVersion = '1.1.51'
        springBootVersion = '2.0.0.BUILD-SNAPSHOT'
    }
    repositories {
        mavenCentral()
        maven { url "https://repo.spring.io/snapshot" }
        maven { url "https://repo.spring.io/milestone" }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:$
        {springBootVersion}")
```

```
        classpath("org.jetbrains.kotlin:kotlin-gradle-plugin:${kotlinVersion}")
        classpath("org.jetbrains.kotlin:kotlin-allopen:${kotlinVersion}")
    }
}

apply plugin: 'kotlin'
apply plugin: 'kotlin-spring'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'

group = 'com.easy.kotlin'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8
compileKotlin {
    kotlinOptions.jvmTarget = "1.8"
}
compileTestKotlin {
    kotlinOptions.jvmTarget = "1.8"
}

repositories {
    mavenCentral()
    maven { url "https://repo.spring.io/snapshot" }
    maven { url "https://repo.spring.io/milestone" }
}

dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
    compile("org.jetbrains.kotlin:kotlin-stdlib-jre8:${kotlinVersion}")
    compile("org.jetbrains.kotlin:kotlin-reflect:${kotlinVersion}")
    testCompile('org.springframework.boot:spring-boot-starter-test')
}
```

其中，spring-boot-gradle-plugin 是 Spring Boot 集成 Gradle 的插件；kotlin-gradle-plugin 是 Kotlin 集成 Gradle 的插件；kotlin-allopen 是 Kotlin 集成 Spring 框架把类全部设置为 open 的插件。

因为 Kotlin 的所有类及其成员默认情况下都是 final（不可继承）的，也就是说你想要继承一个类，就要不断地写各种修饰符来打开类为可继承的。而使用 Java 写的 Spring 框架中大量使用了继承和覆写，这个时候使用 kotlin-allopen 插件结合 kotlin-spring 插件，可以自动把 Spring 相关的所有注解的类都设置为 open。

spring-boot-starter-web 就是 Spring Boot 中提供的使用 Spring 框架进行 Web 应用开发的启动器。

kotlin-stdlib-jre8 是 Kotlin 使用 Java 8 的库，kotlin-reflect 是 Kotlin 的反射库。

Spring Boot 项目的整体依赖情况如图 13-4 所示。

可以看出，spring-boot-starter-web 中已经引入了我们所需要的 JSON、Tomcat、Validator、Web MVC（其中引入了 Spring 框架的核心 Web、Context、AOP、Beans、Expressions、Core）等框架。

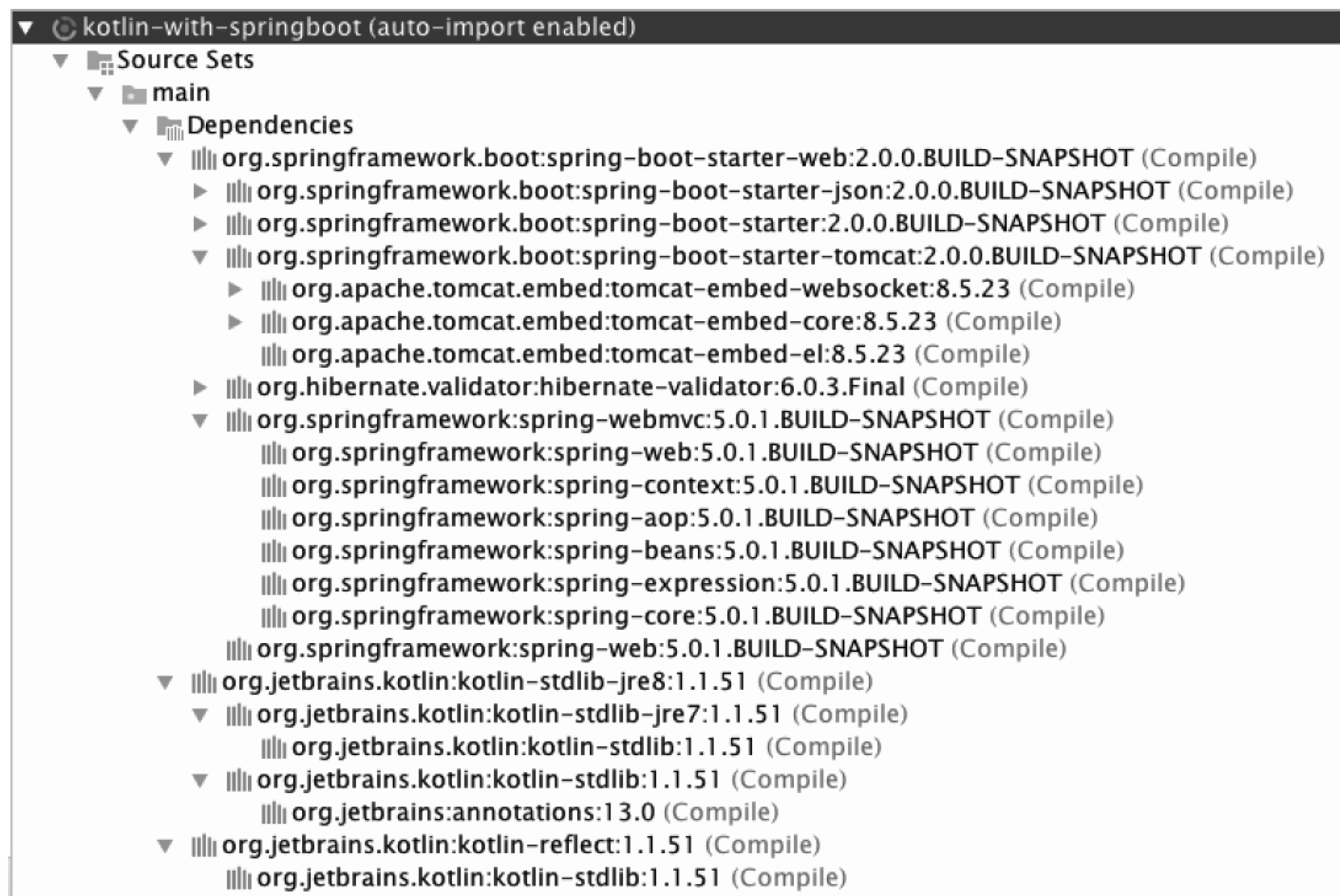


图 13-4　Spring Boot 项目的整体依赖情况

3．Spring Boot项目的入口类 KotlinWithSpringbootApplication

自动生成的 Spring Boot 项目的入口类 KotlinWithSpringbootApplication 如下：

```
package com.easy.kotlin.kotlinwithspringboot

importorg.springframework.boot.SpringApplication
import org.springframework.boot.autoconfigure.SpringBootApplication

@SpringBootApplication
class KotlinWithSpringbootApplication

fun main(args: Array<String>) {
    SpringApplication.run(KotlinWithSpringbootApplication::class.java, *args)
}
```

其中，@SpringBootApplication 注解是 3 个注解的组合，分别是@SpringBootConfiguration 后台使用的@Configuration、@EnableAutoConfiguration 和@ComponentScan。

由于这些注解一般都是一起使用，因此 Spring Boot 提供了这个@SpringBootApplication 统一的注解。该注解的定义源码如下：

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
```

```
@ComponentScan(
    excludeFilters = {@Filter(
    type = FilterType.CUSTOM,
    classes = {TypeExcludeFilter.class}
), @Filter(
    type = FilterType.CUSTOM,
    classes = {AutoConfigurationExcludeFilter.class}
)}
)
public @interface SpringBootApplication {
    ...
}
```

main() 函数中的 KotlinWithSpringbootApplication::class.java 是一个使用反射获取 KotlinWithSpringbootApplication 类的 Java Class 引用。这也正是我们在依赖中引入 kotlin-reflect 包的用途所在。

4. 写Hello World控制器

下面我们来实现一个简单的Hello World控制器。首先新建 HelloWorldController Kotlin 类，代码实现如下：

```
package com.easy.kotlin.kotlinwithspringboot

import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.ResponseBody

@Controller
class HelloWorldController {

    @RequestMapping("/")
    @ResponseBody
    fun home(): String {
        return "Hello World!"
    }

}
```

5. 启动运行

系统默认端口号是 8080，我们在 application.properties 中添加一行服务端口号的配置。

```
server.port=8000
```

然后直接启动入口类 KotlinWithSpringbootApplication，可以看到启动日志。

```
...o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8000 (http)
.e.k.k.KotlinWithSpringbootApplicationKt : Started KotlinWithSpringboot-
ApplicationKt in 7.944
seconds (JVM running for 9.049)
```

也可以选择 IDEA 的 Gradle 工具栏里的 Tasks|application|bootRun 命令来启动程序，如图 13-5 所示。

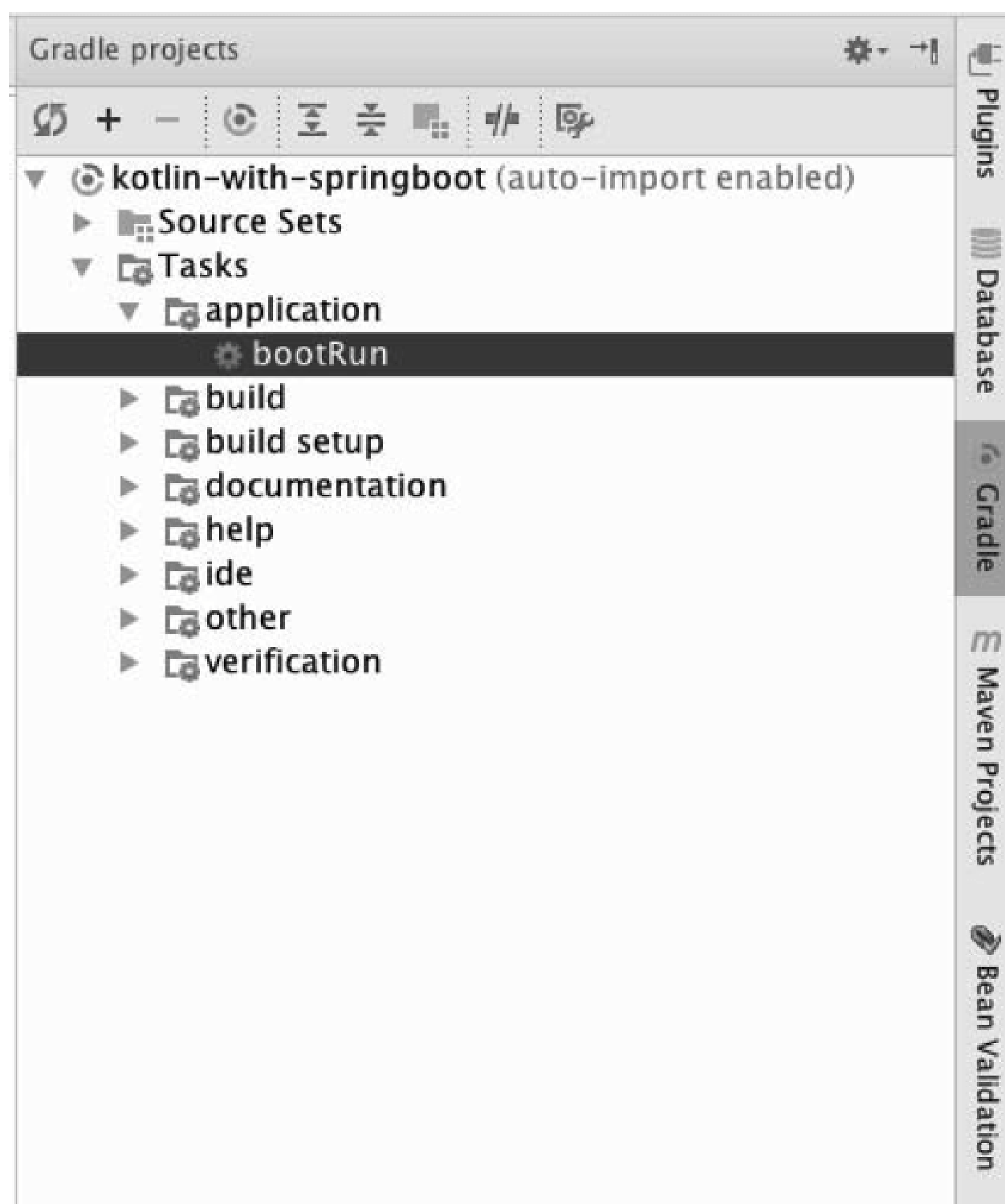


图 13-5　选择 Gradle 的 bootRun

启动完毕后，直接在浏览器中打开 http://127.0.0.1:8000/，可以看到浏览器中输出了 Hello World!，如图 13-6 所示。

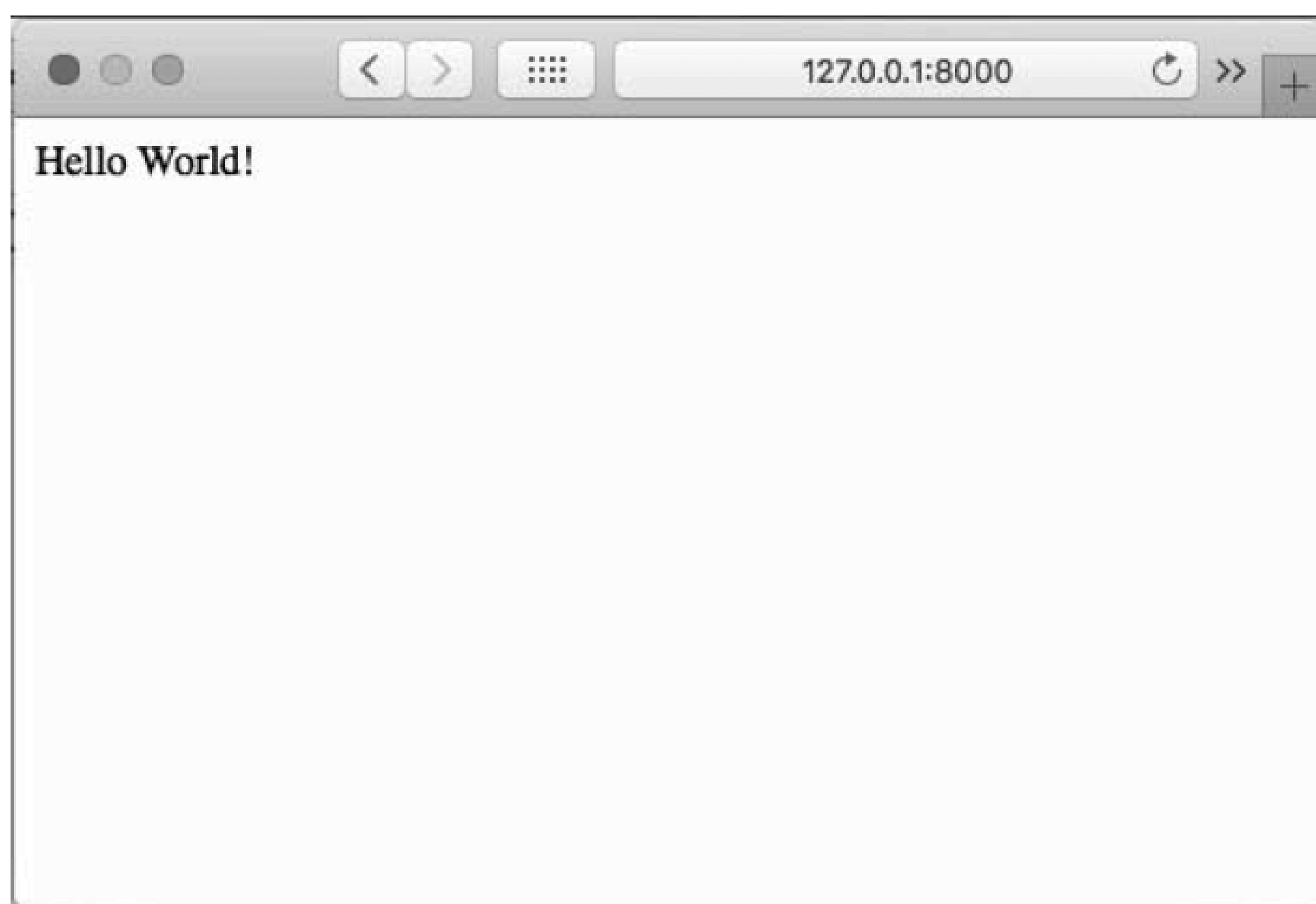


图 13-6　在浏览器中输出 Hello World!

本节项目源码地址是 https://github.com/EasySpringBoot/kotlin-with-springboot。

下面将使用 Kotlin + Spring Boot 框架实现一个简单的图片爬虫的 Web 应用实例。上面我们已经看到了使用 Kotlin 集成 Spring Boot 框架开发的基本步骤。下面将给出一个 Kotlin 集成 Spring Boot 开发框架，使用 MySQL 数据库、Spring Data JPA 框架、Freemarker 模板引擎的完整 Web 项目的实例。首先我们来简单介绍一下系统的技术栈。

13.2　系统功能与技术栈

本节介绍使用 Kotlin 集成 Spring Boot 框架开发一个完整的图片爬虫 Web 应用，基本功能如下：

- 定时抓取图片搜索 API 根据关键字搜索返回的图片 JSON 信息，解析入库；
- Web 页面分页展示图片列表，支持收藏、删除等功能；
- 列表支持根据图片分类进行模糊搜索。

涉及的主要技术栈有如下几种。

- 编程语言：Kotlin；
- 数据库层：MySQL、mysql-jdbc-driver、JPA；
- 企业级开发框架：Spring Boot、Spring MVC；
- 视图层模板引擎：Freemarker；
- 前端框架：jQuery、Bootstrap、Bootstrap-table；
- 工程构建工具：Gradle。

13.3　准 备 工 作

使用 Spring Initializr 创建项目，首先配置项目基本信息和依赖，如图 13-7 所示。

图 13-7　配置项目基本信息和依赖

自动生成项目源码工程，导入 IDEA 中，等构建完毕后将得到下面的工程目录：

```
picture-crawler$ tree
.
├── build.gradle
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── picture-crawler.iml
└── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               └── picturecrawler
    │   │                   └── PictureCrawlerApplication.kt
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        ├── java
        ├── kotlin
        │   └── com
        │       └── easy
        │           └── kotlin
        │               └── picturecrawler
        │                   └── PictureCrawlerApplicationTests.kt
        └── resources

21 directories, 9 files
```

自动生成的 build.gradle 文件如下：

```
buildscript {
    ext {
        kotlinVersion = '1.1.51'
        springBootVersion = '2.0.0.BUILD-SNAPSHOT'
    }
    repositories {
        mavenCentral()
        maven { url "https://repo.spring.io/snapshot" }
        maven { url "https://repo.spring.io/milestone" }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:$
        {springBootVersion}")
        classpath("org.jetbrains.kotlin:kotlin-gradle-plugin:${kotlinVersion}")
        classpath("org.jetbrains.kotlin:kotlin-allopen:${kotlinVersion}")
    }
}

apply plugin: 'kotlin'
apply plugin: 'kotlin-spring'
apply plugin: 'eclipse
'apply plugin: 'org.springframework.boot'
```

```
apply plugin: 'io.spring.dependency-management'

group = 'com.easy.kotlin'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8
compileKotlin {
    kotlinOptions.jvmTarget = "1.8"
}
compileTestKotlin {
    kotlinOptions.jvmTarget = "1.8"
}

repositories {
    mavenCentral()
    maven { url "https://repo.spring.io/snapshot" }
    maven { url "https://repo.spring.io/milestone" }
}

dependencies {
    compile('org.springframework.boot:spring-boot-starter-freemarker')
    compile('org.springframework.boot:spring-boot-starter-data-jpa')
    compile('org.springframework.boot:spring-boot-starter-quartz')
    compile('org.springframework.boot:spring-boot-starter-web')
    compile("org.jetbrains.kotlin:kotlin-stdlib-jre8:${kotlinVersion}")
    compile("org.jetbrains.kotlin:kotlin-reflect:${kotlinVersion}")
    runtime('mysql:mysql-connector-java')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}
```

可以看到，在 build.gradle 中新增了 spring-boot-starter-freemarker、mybatis-spring-boot-starter、spring-boot-starter-quartz、mysql-connector-java 等依赖。在这些 starter 中已经封装了这个工具链所需要的依赖库。整个项目的依赖情况如图 13-8 所示。

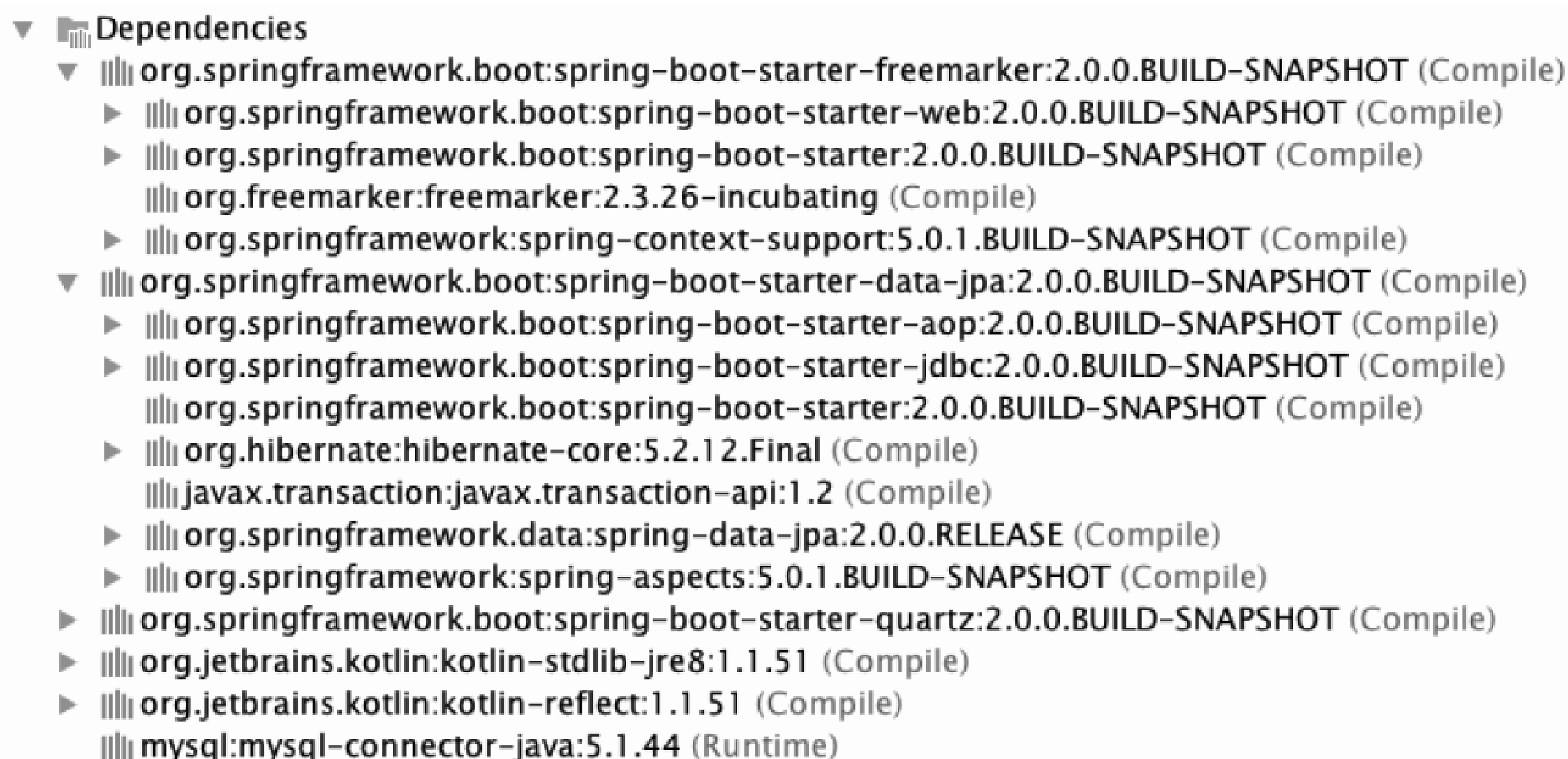


图 13-8　整个项目的依赖情况

目前我们的工程已经具备了连接 MySQL 数据库、解析 Freemarker 的.ftl 模板文件等能力了。但是此时如果启动会报错：

```
BeanCreationException: Error creating bean with name 'dataSource' defined
in class path
resource [org/springframework/boot/autoconfigure/jdbc/ DataSourceConfiguration-
$Hikari.class]
```

创建 dataSource Bean 失败。因为我们还没有配置任何数据库连接信息。下面我们来配置数据源 dataSource。

13.4 配置数据层

Spring Boot 的数据源配置在 application.properties 中是以 spring.datasource 为前缀的。例如，新建一个 wotu 库：

```
CREATE SCHEMA 'wotu' DEFAULT CHARACTER SET utf8 ;
```

配置数据库的链接 URL、用户名、密码信息如下：

```
spring.datasource.url=jdbc:mysql://localhost:3306/wotu?zeroDateTimeBeha
vior=convertToNull&characterEncoding=utf8&characterSetResults=utf8&useS
SL=false
spring.datasource.username=root
spring.datasource.password=root

spring.datasource.testWhileIdle=true
spring.datasource.validationQuery=SELECT 1
```

然后再次启动应用，可以发现成功启动了。

13.5 数据持久层开发

下面我们从数据持久层开始构建应用。首先来设计数据库的表结构。

13.5.1 数据库表结构

首先设计数据库的表结构如下：

```
CREATE TABLE 'picture' (
  'id' bigint(20) NOT NULL AUTO_INCREMENT,
  'category' varchar(255) DEFAULT NULL,
  'deleted_date' datetime DEFAULT NULL,
  'gmt_created' datetime DEFAULT NULL,
  'gmt_modified' datetime DEFAULT NULL,
  'is_deleted' int(11) NOT NULL,
  'url' varchar(500) NOT NULL,
  'version' int(11) NOT NULL,
  'is_favorite' int(11) NOT NULL,
  PRIMARY KEY ('id','url'),
  KEY 'url' ('id','url') USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

因为我们使用的是 JPA，所以只需要写好实体类代码，启动应用即可在 MySQL 数据库中自动创建表结构。实体类代码如下：

```
package com.easy.kotlin.picturecrawler.entity

import java.util.*
import javax.persistence.*

@Entity
class Image {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    var id: Long = -1
    @Version
    var version: Int = 0
    var category: String = ""
    var isFavorite: Int = 0
    var url: String = ""
    var gmtCreated: Date = Date()
    var gmtModified: Date = Date()
    var isDeleted: Int = 0  //1 Yes,0 No
    var deletedDate: Date = Date()

    override fun toString(): String {
        return "Image(id=$id, version=$version, category='$category', isFavorite=
        $isFavorite,url='$url',gmtCreated=$gmtCreated,gmtModified=$gmtModified,
         isDeleted=$isDeleted,deletedDate=$deletedDate)"
    }
}
```

13.5.2　配置 JPA

下面再配置一下 JPA 的一些行为：

```
spring.jpa.database=MYSQL
spring.jpa.show-sql=true
# Hibernate ddl auto (create, create-drop, update)
spring.jpa.hibernate.ddl-auto=update
# Naming strategy
spring.jpa.hibernate.naming-strategy=org.hibernate.cfg.ImprovedNamingSt
rategy
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dia
lect
```

1．ddl-auto的值

其中，spring.jpa.hibernate.ddl-auto 的值有：create、create-drop、update、validate、none，如表 13-1 中分别进行了简单的说明。

表 13-1　ddl-auto 的值说明

值	说　明
create	每次加载 hibernate 会自动创建表，以后启动会覆盖之前的表。所以这个值基本不用，因为严重的情况下会导致数据丢失
create-drop	每次加载 hibernate 时根据 model 类生成表，但是 sessionFactory 一关闭表就自动删除，下一次启动会重新创建
update	加载 hibernate 时根据实体类 model 创建数据库表，这时表名的依据是@Entity 注解的值或者@Table 注解的值。sessionFactory 关闭表不会删除，且下一次启动会根据实体 model 更新结构或者有新的实体类时创建新的表

续表

值	说　明
validate	启动时验证表的结构，不会创建表
none	启动时不做任何操作

一般在开发项目的过程中，通常会选用 update 选项。

再次启动应用，启动完毕后可以看到数据库中已经自动创建了 image 表，如图 13-9 所示。

	Field	Type	Null	Key	Default	Extra
1	id	bigint(20)	NO	PRI	<null>	auto_increment
2	category	varchar(255)	YES	MUL	<null>	
3	deleted_date	datetime	YES		<null>	
4	gmt_created	datetime	YES		<null>	
5	gmt_modified	datetime	YES		<null>	
6	is_deleted	int(11)	NO		<null>	
7	is_favorite	int(11)	NO		<null>	
8	url	varchar(255)	YES	UNI	<null>	
9	version	int(11)	NO		<null>	

图 13-9　数据库中已经自动创建的 image 表

2. 声明数据表的索引

为了达到更高的性能，我们建立类别 category 字段和 url 索引，其中 url 是唯一索引。

```
ALTER TABLE 'sotu'.'image'
  ADD INDEX 'idx_category' ('category' ASC),
  ADD UNIQUE INDEX 'uk_url' ('url' ASC);
```

而实际上不需要手工写上面的 SQL 代码然后再去数据库中执行，只需要写下面的实体类：

```
package com.easy.kotlin.picturecrawler.entity

import java.util.*
import javax.persistence.*

@Entity
@Table(indexes = arrayOf(
        Index(name = "idx_url", unique = true, columnList = "url"),
        Index(name = "idx_category", unique = false, columnList = "category")))
class Image {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    var id: Long = -1
    @Version
    var version: Int = 0

    @Column(length = 255, unique = true, nullable = false)
    var category: String = ""
    var isFavorite: Int = 0

    @Column(length = 255, unique = true, nullable = false)
    var url: String = ""
```

```
    var gmtCreated: Date = Date()
    var gmtModified: Date = Date()
    var isDeleted: Int = 0  //1 Yes, 0 No
    var deletedDate: Date = Date()

    override fun toString(): String {
        return "Image(id=$id, version=$version, category='$category', isFavorite=
       $isFavorite, url='$url', gmtCreated=$gmtCreated, gmtModified=
       $gmtModified, isDeleted=$isDeleted, deletedDate=$deletedDate)"
    }
}
```

然后在@Table 注解里指定为 url、category 建立索引，以及设定 url 唯一性约束 unique=true。

```
@Table(indexes = arrayOf(
        Index(name = "idx_url", unique = true, columnList = "url"),
        Index(name = "idx_category", unique = false, columnList = "category")))
```

启动应用的时候，JPA 会去解析我们的注解生成对应的 SQL，并且自动去执行相应的 SQL。例如，字段 url 的唯一索引约束，我们可以在启动日志中看到如下输出：

```
Hibernate: alter table image drop index idx url
Hibernate: alter table image add constraint idx_url unique (url)
```

其中，Index 是@Index 注解，作为参数使用的时候不需要加@。我们再举个例子，实体类代码如下：

```
package com.easy.kotlin.picturecrawler.entity

import java.util.*
import javax.persistence.*

@Entity
@Table(indexes = arrayOf(Index(name = "idx key word", columnList = "keyWord",
unique = true)))
class SearchKeyWord {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    var id: Long = -1
    @Column(length = 50, unique = true, nullable = false)
    var keyWord: String = ""
    var gmtCreated: Date = Date()
    var gmtModified: Date = Date()
    var isDeleted: Int = 0  //1 Yes, 0 No
    var deletedDate: Date = Date()
}
```

重启应用，可以看到 Hibernate 日志如下：

```
Hibernate: create table search key word (id bigint not null auto increment,
deleted date  datetime,  gmt created  datetime,  gmt modified  datetime,
is deleted integer not null, key word varchar(50) not null, primary key (id))
engine=MyISAM
Hibernate: alter table search_key_word drop index UK_lvmjkr0dkesio7a33ejre5c26
Hibernate: alter table search_key_word add constraint UK_lvmjkr0dkesio7a33ej-
re5c26 unique (key_word)
```

自动生成的表结构如图 13-10 所示。

	Field	Type	Null	Key	Default	Extra
1	id	bigint(20)	NO	PRI	<null>	auto_increment
2	deleted_date	datetime	YES		<null>	
3	gmt_created	datetime	YES		<null>	
4	gmt_modified	datetime	YES		<null>	
5	is_deleted	int(11)	NO		<null>	
6	key_word	varchar(50)	NO	UNI	<null>	

图 13-10　自动生成的表结构

其中，@Column(length=50,unique=true, nullable=false)这一句指定了 keyWord 字段的长度是 50，有唯一约束，不可空。对应生成的数据库表字段 key_word 信息中：Type 是 varchar(50)，Null 是 NO，Key 是唯一键 UNI。

3. 主键自动生成策略

我们使用@Id 注解来标注主键字段：

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
var id: Long = -1
```

其中的@GeneratedValue(strategy=GenerationType.IDENTITY)注解需要重点介绍一下。这里的 GenerationType 是主键 ID 的生成规则。JPA 提供的 4 种标准用法为 TABLE、SEQUENCE、IDENTITY、AUTO，每个值的说明如表 13-2 所示。

表 13-2　GenerationType的值说明

GenerationType	说　明
TABLE	使用一个特定的数据库表格来保存主键
SEQUENCE	根据底层数据库的序列来生成主键，条件是数据库支持序列
IDENTITY	主键由数据库自动生成（主要是自动增长型）
AUTO	主键由程序控制

设计源码目录如下：

```
├── src
│   ├── main
│   │   ├── java
│   │   ├── kotlin
│   │   │   └── com
│   │   │       └── easy
│   │   │           └── kotlin
│   │   │               └── picturecrawler
│   │   │                   ├── PictureCrawlerApplication.kt
│   │   │                   ├── controller
│   │   │                   ├── dao
│   │   │                   ├── entity
│   │   │                   ├── job
│   │   │                   └── service
...
```

其中，controller 包下面放置 Controller 控制器代码；entity 包下面放置对应到数据库表的实体类代码；dao 包下面放置数据访问层逻辑代码；service 包下面放置业务逻辑实现代码；job 包下面放置定时任务代码。

13.6　JSON 数据解析

我们的图片搜索 API 返回的数据结构是 JSON 格式的，内容示例如下：


```
{
  "queryEnc": "%E7%BE%8E%E5%A5%B3",
  "queryExt": "美女",
  "listNum": 3900,
  "displayNum": 415337,
  "gsm": "5a",
  "bdFmtDispNum": "约 415,000",
  "bdSearchTime": "",
  "isNeedAsyncRequest": 1,
  "bdIsClustered": "1",
  "data": [
    {
      "adType": "0",
      "hasAspData": "0",
      "thumbURL":
      "http://img5.imgtn.bdimg.com/it/u=2817128514,340025963&fm=27&gp=0.jpg",
      "middleURL":
      "http://img5.imgtn.bdimg.com/it/u=2817128514,340025963&fm=27&gp=0.jpg",
      "largeTnImageUrl": "",
      "hasLarge": 0,
       ...
      "currentIndex": "",
      "width": 800,
      "height": 958,
      "type": "jpg",
      "is gif": 0,
       ...
      "bdImgnewsDate": "1970-01-01 08:00",
      "fromPageTitle": "",
      "fromPageTitleEnc": "性感美女",
    ...
}
```


我们只需要取出其中的 thumbURL 和 fromPageTitleEnc 两个字段的值。我们使用 fastjson 来解析这个 JSON 字符串。

```
try {
    val obj = JSON.parse(jsonstr) as Map<*, *> //(1) parse()函数
    val dataArray = obj.get("data") as JSONArray //(2) 取出 data 转换成 JSONArray
    dataArray.forEach {
        val category = (it as Map<*, *>).get("fromPageTitleEnc") as String
                                          //(3) 获取目标 key
        val url = it.get("thumbURL") as String //(4)
        val imageResult = ImageCategoryAndUrl(category = category, url = url)
        imageResultList.add(imageResult)
    }
} catch (ex: Exception) {
    ...
}
```

代码说明如下：

- 使用 fastjson 的 JSON 类的 parse()方法来解析 JSON 字符串 jsonstr，然后把类型转换成 Map<*,*>，这里的 * 是泛型星号通配符。
- 取出 Map 中 key 为 data 的值，它是一个数组。然后再次使用 as 操作符把它转成 JSONArray。
- 遍历 dataArray 中的元素 it，转换成 Map<*,*> 类型，然后取出相应字段的值。
- 由于在第三步中已经将 it 转换成了 Map<*,*>类型，编译器已经记住了 it 转换后的类型，所以在这里的 it 类型已经是 Map 类型了，我们可以直接当作 Map 来使用。

其中的 ImageCategoryAndUrl 对象是我们定义的数据转换对象。

```
data class ImageCategoryAndUrl(val category: String, val url: String)
```

搜索图片的 APiBuilder 如下：

```
object ImageSearchApiBuilder {
    fun build(word: String, page: Int): String {
        return
"http://image.baidu.com/search/acjson?tn=resultjson com&ipn=rj&fp=resul
t&word=${word}&pn=${30 * page}&rn=30"
    }
}
```

我们来写个单元测试：

```
package com.easy.kotlin.picturecrawler

import com.easy.kotlin.picturecrawler.api.ImageSearchApiBuilder

import com.easy.kotlin.picturecrawler.service.JsonResultProcessor
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4::class)
class JsonResultProcessorTest {
    @Test
    fun testJsonResultProcessor() {
        val list = JsonResultProcessor.getImageCategoryAndUrlList(ImageSearchApi-
         Builder.build("美女", 1))
        println(list)
    }
}
```

输出如下：

```
[ImageCategoryAndUrl(category=美女写真集,
url=http://img1.imgtn.bdimg.com/ it/u=3772875022,724775083&fm=27&gp=0.jpg),
ImageCategoryAndUrl(category=美女写真 美女_美女写真 美女_美女写真 美女,
url=http://img0.imgtn.bdimg.com/it/u= 3312193685,1215837845&fm=11&gp=0.jpg),
ImageCategoryAndUrl(category=...
```

13.7 数据入库逻辑实现

现在我们已经有了数据的表结构和实体类代码，同时也已经有业务源数据了。我们现

在要做的是把爬到的图片信息存储到数据库中，同时，重复的 url 信息不再重复存储。

新建一个实现 PagingAndSortingRepository 的 ImageRepository 接口如下：

```
interface ImageRepository : PagingAndSortingRepository<Image, Long>
```

只要上面一行代码，就可以直接使用 ImageRepository 的 CRUD（增加、修改、删除、查询）方法了。因为 JPA 框架会自动生成这些方法。下面代码中的 PagingAndSorting Repository 是带分页功能的，它继承了 CrudRepository 接口：

```
@NoRepositoryBean
public interface PagingAndSortingRepository<T, ID> extends CrudRepository
<T, ID> {
    Iterable<T> findAll(Sort sort);
    Page<T> findAll(Pageable pageable);
}
```

而在接口 CrudRepository 中定义了我们能够直接使用的 CRUD 方法。

```
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);
    <S extends T> Iterable<S> saveAll(Iterable<S> entities);
    Optional<T> findById(ID id);
    boolean existsById(ID id);
    Iterable<T> findAll();
    Iterable<T> findAllById(Iterable<ID> ids);
    long count();
    void deleteById(ID id);
    void delete(T entity);
    void deleteAll(Iterable<? extends T> entities);
    void deleteAll();
}
```

我们入库就直接使用 save()(Sentity)方法。但是为了保证重复的 url 不再被保存，需要写个函数来判断在数据库中是否存在当前 URL。我们直接使用 selectcount()语句来判断即可，当且仅当 selectcount()出来的值等于 0（表明数据库中不存在此 url）时，才进行入库动作。在 ImageRepository 接口中直接声明函数即可，代码如下：

```
@Query("select count(*) from #{#entityName} a where a.url = :url")
fun countByUrl(@Param("url") url: String): Int
```

入库逻辑代码如下：

```
if (imageRepository.countByUrl(url) == 0) {
    val Image = Image()
    Image.category = category
    Image.url = url
    imageRepository.save(Image)
}
```

13.8　定时调度任务

为了简单起见，我们直接使用 Spring 自带的 scheduling 包下的@Schedules 注解来实现

任务的定时执行。需要注意的是，要在 Spring Boot 的启动类上面添加注解如下：

```
@SpringBootApplication
@EnableScheduling
class PictureCrawlerApplication
```

我们的定时任务代码如下：

```
package com.easy.kotlin.picturecrawler.job

import com.easy.kotlin.picturecrawler.service.CrawImageService
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.scheduling.annotation.Scheduled
import org.springframework.stereotype.Component
import java.util.*

@Component
class ImageCrawlerJob {

    @Autowired lateinit var CrawImagesService: CrawImageService
    @Scheduled(cron = "0 */5 * * * ?")
    fun job() {
        println("开始执行定时任务: ${Date()}")
        CrawImagesService.doCrawJob()
    }
}
```

其中，@Scheduled(cron="0*/5***?")表示每隔 5 分钟执行一次图片的抓取任务。然后重新启动应用，会看到每隔 5 分钟，定时任务会运行一次。

到目前为止，我们的原始数据已经入库。下面我们将要进行控制器层代码和视图展示层模板引擎代码开发，最后是前端页面展示部分的代码开发。

13.9 HTTP 接口开发

本节介绍服务端的 HTTP 接口的开发，通过使用 Spring Data JPA 提供的分页功能来实现一个分页查询的后端 HTTP 接口。

13.9.1 实现分页查询接口

下面实现一个分页查询 HTTP 接口，这个接口是 http://127.0.0.1:8000/sotuJson?page=10&size=3，该接口返回的数据是 JSON 格式，代码如下：

```
{
  "content": [
    {
      "id": 5981,
      "version": 0,
      "category": "南非,动物世界,非洲地区旅游景点,风景名胜",
      "url": "http://img0.imgtn.bdimg.com/it/u=2871771810,3599000038&fm=
     27&gp=0.jpg",
      "gmtCreated": 1508858697000,
```

```
    "gmtModified": 1508858697000,
    "deletedDate": 1508858697000
   },
  ...
 ],
 "pageable": {
   "sort": {
     "sorted": true,
     "unsorted": false
   },
   "offset": 30,
   "pageSize": 3,
   "pageNumber": 10,
   "paged": true,
   "unpaged": false
 },
 "last": false,
 "totalPages": 2004,
 "totalElements": 6011,
 "size": 3,
 "numberOfElements": 3,
 "sort": {
   "sorted": true,
   "unsorted": false
 },
 "number": 10,
 "first": false
}
```

13.9.2　@Query 注解与 #{#entityName}

在 Spring Data JPA 中提供了基本的 CRUD 操作、分页查询、排序等。我们先来实现 ImageRepository 接口中的 findAll()函数：

```
@Query("SELECT a from #{#entityName} a where a.isDeleted=0 order by a.id desc")
override fun findAll(pageable: Pageable): Page<Image>
```

其中，@Query 是 JPA 中的查询注解。

JPA 中可以执行两种方式的查询，一种是使用 JPQL，另一种是使用 Native SQL。其中，JPQL 是基于 Entity 对象（@Entity 注解标注的对象）的查询，可以消除不同数据库间 SQL 语句上的差异；本地 SQL 语句是基于传统的 SQL 查询，是对 JPQL 查询语句的补充。

在 JPQL 中可以使用 SPEL 表达式#{#entityName}代替本来实体的名称，而 Spring Data JPA 会自动根据 Image 实体上对应的@Entity(name="Image")或者是默认的@Entity，自动将实体名称填入 HQL 语句中。

实体类 Image 使用@Entity 注解后，Spring Data JPA 的 EntityManager 会将实体类 Image 纳入管理。默认的 #{#entityName} 的值就是 Image，如果指定其中的@Entity(name="Image")name 的值，那么#{#entityName}就是指定的值。

在 JPQL 语句中：

```
SELECT a from #{#entityName} a where a.isDeleted=0 order by a.id desc
```

我们就可以像访问 Kotlin 类属性一样来访问字段值。注意，这里的 a.isDeleted 是属性名称。

13.9.3 Pageable 与 Page

本节介绍实现分页的两个关键类型：Pageable 与 Page。

1. 分页方法传入的Pageable参数

Spring Data JPA 的 PagingAndSortingRepository 接口已经提供了对分页的支持，查询的时候我们只需要传入一个 Pageable 类型的实现类。这个 Pageable 接口定义如下：

```
package org.springframework.data.domain;
import java.util.Optional;
import org.springframework.util.Assert;

public interface Pageable {
    static Pageable unpaged() {
        return Unpaged.INSTANCE;
    }

    default boolean isPaged() {
        return true;
    }

    default boolean isUnpaged() {
        return !isPaged();
    }

    int getPageNumber();

    int getPageSize();

    long getOffset();

    Sort getSort();

    default Sort getSortOr(Sort sort) {
        Assert.notNull(sort, "Fallback Sort must not be null!");
        return getSort().isSorted() ? getSort() : sort;
    }

    Pageable next();

    Pageable previousOrFirst();

    Pageable first();

    boolean hasPrevious();

    default Optional<Pageable> toOptional() {
        return isUnpaged() ? Optional.empty() : Optional.of(this);
    }
}
```

Spring Data JPA 中提供的 PageRequest 类已经实现了 Pageable 接口，可以像下面这样直接使用：

```
val sort = Sort(Sort.Direction.DESC, "id")
```

```
val pageable = PageRequest.of(page, size, sort)
```

其中，Direction 是 Sort 类中定义的注解：

```
public static enum Direction {
        ASC, DESC;
}
```

Sort 类的构造函数签名如下：

```
public Sort(Direction direction, String... properties) {
    this(direction, properties == null ? new ArrayList<>() : Arrays.asList
    (properties));
}
```

这里 Sort(Sort.Direction.DESC,"id")传入的是根据 ID 进行降序排序。

2. 分页返回类型Page

findAll()函数的返回类型是 Page，这里的 Page 类型是 Spring Data JPA 分页结果的返回对象，Page 继承了 Slice。Page 和 Slice 这两个接口的定义如下:

```
public interface Page<T> extends Slice<T> {
    static <T> Page<T> empty() {
        return empty(Pageable.unpaged());
    }
    static <T> Page<T> empty(Pageable pageable) {
        return new PageImpl<>(Collections.emptyList(), pageable, 0);
    }
    int getTotalPages();
    long getTotalElements();
    <U> Page<U> map(Function<? super T, ? extends U> converter);
}

public interface Slice<T> extends Streamable<T> {
    int getNumber();
    int getSize();
    int getNumberOfElements();
    List<T> getContent();
    boolean hasContent();
    Sort getSort();
    boolean isFirst();
    boolean isLast();
    boolean hasNext();
    boolean hasPrevious();
    default Pageable getPageable() {
        return PageRequest.of(getNumber(), getSize(), getSort());
    }
    Pageable nextPageable();
    Pageable previousPageable();
    <U> Slice<U> map(Function<? super T, ? extends U> converter);
}
```

这个分页对象 Pageable 的数据结构信息足够我们在前端实现分页交互页面时使用。下面来实现分页查询所有 image 表记录的 REST API 接口。在 controller 包路径下新建 ImageController 类，在该类中使用@Controller 注解。

```
package com.easy.kotlin.picturecrawler.controller

import com.easy.kotlin.picturecrawler.dao.ImageRepository
import com.easy.kotlin.picturecrawler.entity.Image
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.data.domain.Page
import org.springframework.data.domain.PageRequest
import org.springframework.data.domain.Sort
import org.springframework.stereotype.Controller
import org.springframework.ui.Model
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RequestMethod
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.ResponseBody
import org.springframework.web.servlet.ModelAndView
import javax.servlet.http.HttpServletRequest

@Controller
class ImageController {
    @Autowired
    lateinit var imageRepository: ImageRepository

    @RequestMapping(value = "sotuJson", method = arrayOf(RequestMethod.GET))
    @ResponseBody
    fun sotuJson(@RequestParam(value = "page", defaultValue = "0") page: Int,
                 @RequestParam(value = "size", defaultValue = "10") size: Int):
                 Page<Image> {
        return getPageResult(page, size)
    }
    private fun getPageResult(page: Int, size: Int): Page<Image> {
        val sort = Sort(Sort.Direction.DESC, "id")
        val pageable = PageRequest.of(page, size, sort)
        return imageRepository.findAll(pageable)
    }
    ...
}
```

其中：

```
@Autowired
lateinit var imageRepository: ImageRepository
```

里使用 lateinit 关键字来修饰我们需要装配的 Bean，表示 imageRepository 延迟初始化。

从上面的代码中可以看出，Kotlin 使用 Spring MVC 框架非常自然，与使用原生 Java 代码几乎一样顺畅。但是有个较明显的区别是 method = arrayOf(RequestMethod.GET)，这里 Kotlin 数组声明的语法是使用 arrayOf()，而在 Java 中只需要使用括号“{}”括起来即可。关于注解中使用数据的语法，在 Kotlin 1.2 版本中可以直接使用方括号“[]”括起来。类似下面这样：

```
import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.GetMapping

@Controller
class IndexController {
    @GetMapping(value = ["", "/", "/index"]) //Kotlin 1.2 中的新特性：注解中
                                                的数组语法
```

```
    fun index(): String {
        return "/index"
    }
}
```

重新运行应用，通过浏览器访问 http://127.0.0.1:8000/sotuJson?page=10&size=3，将看到分页对象 Page 的 JSON 字符串格式的输出结果。

3. 模糊搜索分页接口实现

下面来实现根据 category 字段的值进行模糊搜索的接口，并同时支持分页。代码如下：

```
@Query("SELECT a from #{#entityName} a where a.isDeleted=0 and a.category like
%:searchText% order by a.id desc")
fun search(@Param("searchText") searchText: String, pageable: Pageable):
Page<Image>
```

其中，@Param("searchText")searchText:String 是搜索关键字参数@Param 注解指定了 JPQL 中的参数名 searchText，对应到 JPQL 中的参数占位符写作:searchText，我们注意到这里模糊查询的语法是：

```
like %:searchText%
```

对应 Controller 中的方法是：

```
@RequestMapping(value = "sotuSearchJson", method = arrayOf(RequestMethod.GET))
@ResponseBody
fun sotuSearchJson(@RequestParam(value = "page", defaultValue = "0") page: Int,
@RequestParam(value = "size", defaultValue = "10") size: Int, @RequestParam
(value =
"searchText", defaultValue = "") searchText: String): Page<Image> {
    return getPageResult(page, size, searchText)
}

private fun getPageResult(page: Int, size: Int, searchText: String):
Page<Image> {
    val sort = Sort(Sort.Direction.DESC, "id")
    val pageable = PageRequest.of(page, size, sort)
    if (searchText == "") {
        return imageRepository.findAll(pageable)
    } else {
        return imageRepository.search(searchText, pageable)
    }
}
```

这里需要注意的是 PageRequest.of(page,size,sort)page 的取值默认是从 0 开始 。

重新运行程序，通过浏览器访问 http://127.0.0.1:8000/sotuSearchJson?page=10&size=3&searchText=秋天，输出如下：


```
{
  "content": [
    {
      "id": 17443,
      "version": 0,
      "category": "初秋岱庙",
      "url": "http://img0.imgtn.bdimg.com/it/u=64076324,3274882882&fm=27&gp=
```

```
                  0.jpg",
    "gmtCreated": 1508924545000,
    "gmtModified": 1508924545000,
    "deletedDate": 1508924545000
  },
  {
    "id": 17280,
    "version": 0,
    "category": "初秋落叶信纸.doc",
    "url": "http://img5.imgtn.bdimg.com/it/u=256290403,1153099708&fm=27&gp=
                  0.jpg",
    "gmtCreated": 1508924528000,
    "gmtModified": 1508924528000,
    "deletedDate": 1508924528000
  },
  {
    "id": 17130,
    "version": 0,
    "category": "初秋的小花图片 12 张  (天堂图片网)",
    "url": "http://img3.imgtn.bdimg.com/it/u=1333940222,533390017&fm=11&gp=
                  0.jpg",
    "gmtCreated": 1508924510000,
    "gmtModified": 1508924510000,
    "deletedDate": 1508924510000
  }
 ],
 "pageable": {
  "sort": {
    "sorted": true,
    "unsorted": false
  },
  "offset": 30,
  "pageSize": 3,
  "pageNumber": 10,
  "paged": true,
  "unpaged": false
 },
 "last": false,
 "totalElements": 148,
 "totalPages": 50,
 "size": 3,
 "number": 10,
 "numberOfElements": 3,
 "sort": {
  "sorted": true,
  "unsorted": false
 },
 "first": false
}
```


13.10 视图模板开发

后端的数据接口已经开发完毕，下面把这些数据展示到前端页面上。

我们使用的视图层模板引擎是 Freemarker，在 Spring Boot 中使用 Freemarker，只需要加入 spring-boot-starter-freemarker 即可。其中，使用默认的配置目录 src\main\resources\

templates，模板文件以.ftl 为后缀。

13.10.1　前端代码结构

我们将前端依赖的外部库静态资源文件全部放到 src\main\resources\static\bower_components 文件夹下。这里我们主要使用的是 jquery.js 和 bootstrap.js 文件。另外，后端的分页接口实现前端分页的功能使用 bootstrap-table.js 库来实现。前端模板文件及 JS 源码文件的目录结构如图 13-11 所示。



图 13-11　前端目录结构

1. head.ftl部分

head.ftl 文件是公共文件头部分，代码如下：

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv=content-type content=text/html;charset=utf-8>
    <meta http-equiv=X-UA-Compatible content=IE=Edge>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,
     shrink-to-fit=no">
    <title>搜图</title>
    <link href="bower_components/bootstrap/dist/css/bootstrap-theme.css"
     rel="stylesheet">
    <link href="bower_components/bootstrap-table/src/bootstrap-table.css"
     rel="stylesheet">
    <link href="bower_components/bootstrap/dist/css/bootstrap.css" rel=
     "stylesheet">
    <link href="bower_components/pnotify/src/pnotify.css" rel="stylesheet">
    <link href="app.css" rel="stylesheet">
</head>
<body>
```

2. foot.ftl部分

foot.ftl 部分是公共页面的底部部分，代码如下：

```
<script src="bower_components/jquery/dist/jquery.js"></script>
<script src="bower_components/bootstrap/dist/js/bootstrap.js"></script>
<script src="bower_components/bootstrap-table/src/bootstrap-table.js"></script>
<script src="bower_components/bootstrap-table/src/locale/bootstrap-table-zh-CN.
js"></script>
<script src="bower_components/pnotify/src/pnotify.js"></script>
<script src="app.js"></script>
</body>
</html>
```

3. nav.ftl部分

nav.ftl 是导航栏部分的代码，使用标准的 Bootstrap 样式库来实现：

```
<nav class="navbar navbar-default" role="navigation">
    <div class="container-fluid">
        <div class="navbar-header">
            <a class="navbar-brand" href="#">搜图</a>
        </div>
        <div>
            <ul class="nav navbar-nav">

                <li class='<#if requestURI=="/sotu_view">active</#if>'><a
                href="sotu_view">美图列表</a></li>
                <li  class='<#if  requestURI=="/sotu_favorite_view">active
                 </#if>'><a href="sotu_favorite_view">精选收藏</a>
                <li class='<#if requestURI=="/search_keyword_view">active
                </#if>'><a href="search_keyword_view">搜索关键字</a>
                </li>

                <li class=""><a href="doCrawJob" target="_blank">执行抓取</a></li>

                <li class="dropdown">
                    <a href="#" class="dropdown-toggle" data-toggle="dropdown">
                        Kotlin <b class="caret"></b>
                    </a>
                    <ul class="dropdown-menu">
                        <li><a href="http://www.jianshu.com/nb/12976878" target=
                        "_blank">Kotlin 极简教程</a></li>
                        <li><a href="http://www.jianshu.com/nb/17117730" target=
                       "_blank">Kotlin 项目实战开发</a></li>
                        <li><a href="#">SpringBoot</a></li>
                        <li><a href="#">Java</a></li>
                        <li class="divider"></li>
                        <li><a href="#">Scala</a></li>
                        <li class="divider"></li>
                        <li><a href="#">Groovy</a></li>
                    </ul>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="#">关于</a>
```

```
                </li>
            </ul>
        </div>
    </div>
</nav>
```

其中，以下代码是实现按 Tab 键切换的时候，active 状态跟随的交互：

```
<li class='<#if requestURI=="/sotu_view">active</#if>'>
    <a href="sotu_view">美图列表</a>
</li>
<li class='<#if requestURI=="/sotu_favorite_view">active</#if>'>
    <a href="sotu_favorite_view">精选收藏</a>
</li>
<li class='<#if requestURI=="/search_keyword_view">active</#if>'>
    <a href="search_keyword_view">搜索关键字</a>
</li>
```

requestURI 是后端的 Controller 获取当前请求传给前端页面的。

```
@RequestMapping(value = *arrayOf("/", "sotu_view"), method = arrayOf
(RequestMethod.GET))
fun sotuView(model: Model, request: HttpServletRequest): ModelAndView {
    model.addAttribute("requestURI", request.requestURI)
    return ModelAndView("sotu_view")
}
```

4. 图片列表页面

新建 sotu_view.ftl 为图片列表页面：

```
<#include 'common/head.ftl'>
<#include 'common/nav.ftl'>
<table id="sotu_table"></table>
<#include 'common/foot.ftl'>
<script src="sotu_table.js"></script>
```

对应的 ModelAndView 控制器代码如下：

```
@RequestMapping(value = *arrayOf("/", "sotu_view"), method = arrayOf
(RequestMethod.GET))
fun sotuView(model: Model, request: HttpServletRequest): ModelAndView {
    model.addAttribute("requestURI", request.requestURI)
    return ModelAndView("sotu_view")
}
```

13.10.2　实现后端分页

我们使用表格插件 bootstrap-table.js 来实现分页的前端样式，主要使用其中的 bootstrapTable 函数来完成。该函数定义如下：

```
$.fn.bootstrapTable = function (option)
```

实现分页的 sotu_table.js 代码如下：

```
$(function () {
   $.extend($.fn.bootstrapTable.defaults, $.fn.bootstrapTable.locales['zh-CN'])
```

```
    var searchText = $('.search').find('input').val()
    var columns = []
    columns.push({
        title: '分类',
        field: 'category',
        align: 'center',
        valign: 'middle',
        width: '5%',
        formatter: function (value, row, index) {
            return value
        }
    }, {
        title: '美图',
        field: 'url',
        align: 'center',
        valign: 'middle',
        formatter: function (value, row, index) {
            return "![](" + value + ")"
        }
    }, {
        title: ' 操作',
        field: 'id',
        align: 'center',
        width: '5%',
        formatter: function (value, row, index) {
            var html = ""
            html += "<div onclick='addFavorite(" + value + ")' name='addFavorite'
           id='addFavorite" + value + "' class='btn btn-default'>收藏</div><p>"
            html += "<div onclick='deleteById(" + value + ")' name='delete'
           id='delete" + value + "' class='btn btn-default'>删除</div>"
            return html
        }
    })

    $('#sotu table').bootstrapTable({
        url: 'sotuSearchJson',                         //(1)
        sidePagination: "server",
        queryParamsType: 'page,size',
        contentType: "application/x-www-form-urlencoded",
        method: 'get',
        striped: false,                                //是否显示行间隔色
        buttonsAlign: 'right',
        smartDisplay: true,
        cache: false,//是否使用缓存，默认为 true，所以一般情况下需要设置一下这个属性（*）
        pagination: true,                              //是否显示分页（*）
        paginationLoop: true,
        paginationHAlign: 'right',                     //right, left
        paginationVAlign: 'bottom',                    //bottom, top, both
        paginationDetailHAlign: 'left',                //right, left
        paginationPreText: ' 上一页',
        paginationNextText: '下一页',
        search: true,
        searchText: searchText,
        searchTimeOut: 500,
        searchAlign: 'right',
        searchOnEnterKey: false,
        trimOnSearch: true,
        sortable: true,                                //是否启用排序
        sortOrder: "desc",                             //排序方式
```

```
        sortName: "id",
        pageNumber: 1,                   //初始化加载第一页，默认第一页
        pageSize: 10,                    //每页的记录行数（*）
        pageList: [8, 16, 32, 64, 128],  //可选的每页数据
        totalField: 'totalElements',     //(2)所有记录 count
        dataField: 'content',       // (3) 后端 json 对应的表格 List 数据的 key
        columns: columns,                //(4)表格每列的值
        queryParams: function (params) {
            return {
                size: params.pageSize,
                page: params.pageNumber - 1,
                sortName: params.sortName,
                sortOrder: params.sortOrder,
                searchText: params.searchText
            }
        },
        classes: 'table table-responsive full-width',
    })
})
```

代码说明如下：

第（1）处的 url:'sotuSearchJson'对应的后端查询接口实现代码是：

```
@RequestMapping(value = "sotuSearchJson", method = arrayOf(RequestMethod.GET))
@ResponseBody
fun sotuSearchJson(@RequestParam(value = "page", defaultValue = "0") page: Int,
@RequestParam(value = "size", defaultValue = "10") size: Int,
@RequestParam(value =
"searchText", defaultValue = "") searchText: String): Page<Image> {
    return getPageResult(page, size, searchText)
}

private fun getPageResult(page: Int, size: Int, searchText: String):
Page<Image> {
    val sort = Sort(Sort.Direction.DESC, "id")
    //注意：PageRequest.of(page,size,sort) page 默认是从 0 开始
    val pageable = PageRequest.of(page, size, sort)
    if (searchText == "") {
        return imageRepository.findAll(pageable)
    } else {
        return imageRepository.search(searchText, pageable)
    }
}
```

第（2）处的 totalField:'totalElements'对应的是 Page 中 totalElements 属性的值。

第（3）处的 dataField:'content'对应的是 Page 中的 content 属性的值。

第（4）处的 columns:columns 是对应到 contentList 中的每个元素的对象属性。例如这段代码：

```
var columns = []
    columns.push({
        title: '分类',
        field: 'category',//(1) 对应 Image 实体类的 category 属性
        align: 'center',
        valign: 'middle',
        width: '5%',
        formatter: function (value, row, index) { //(2)
```

```
            return value
        },
        ...
}
```

代码说明如下：

第（1）处的 field:'category'对应的就是 Image 实体类的 category 属性名称。

然后在第（2）处的 formatter:function(value,row,index)函数中处理该单元格显示的 HTML 样式。其中，value 值对应该属性的值，row 中存储的是这一行对象的值，index 是下标。

重新启动程序，将看到分页及模糊搜索的效果，如图 13-12 所示。

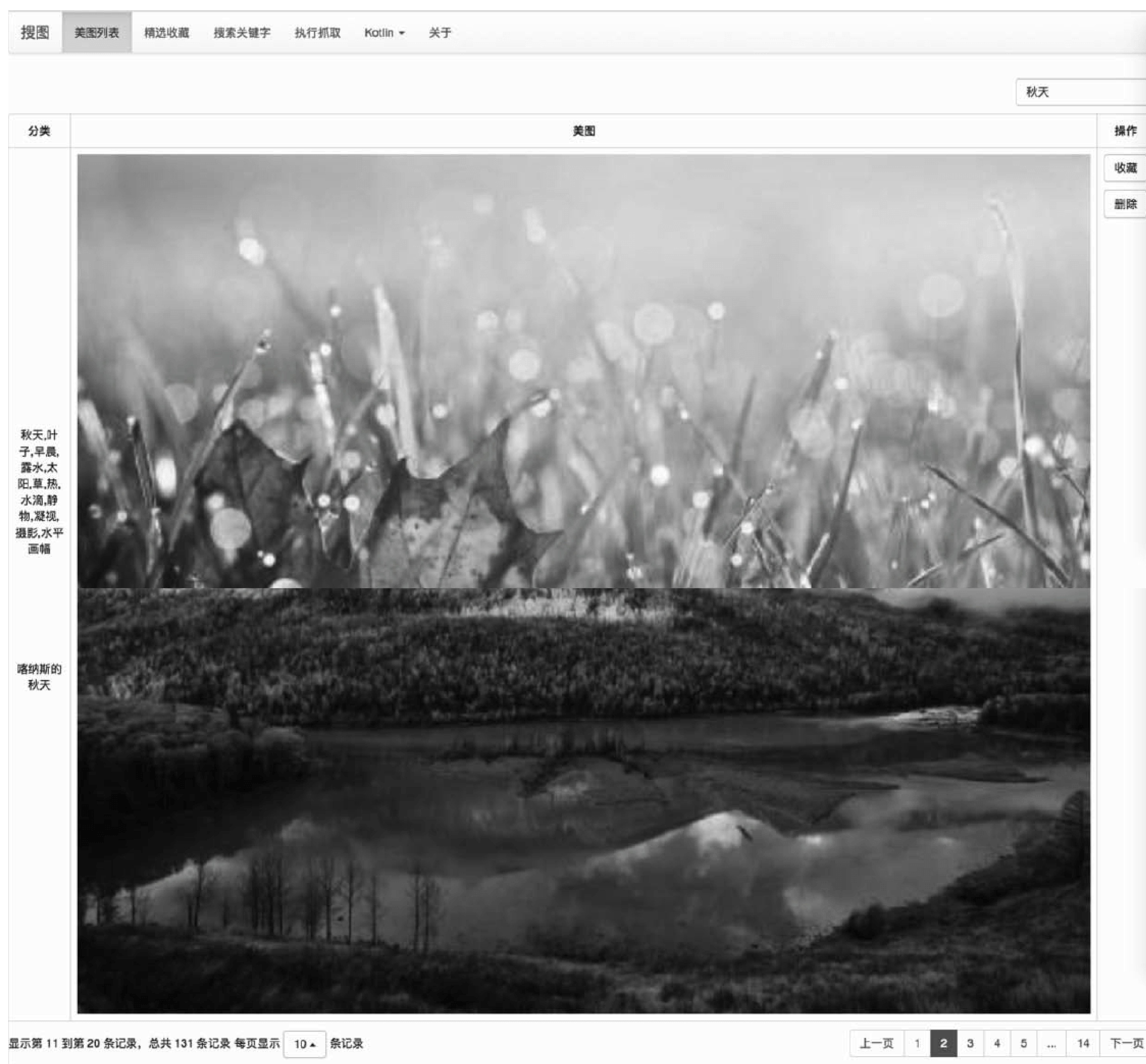


图 13-12　分页及模糊搜索的效果

其中，Bootstrap-table 的完整配置项在 bootstrap-table.js 源码（https://github.com/wenzhixin/bootstrap-table）中的 BootstrapTable.DEFAULTS 这行代码中：

```
BootstrapTable.DEFAULTS = {
      classes: 'table table-hover',
```

```
        sortClass: undefined,
        locale: undefined,
        height: undefined,
        undefinedText: '-',
        sortName: undefined,
        sortOrder: 'asc',
        sortStable: false,
        rememberOrder: false,
        striped: false,
        columns: [[]],
        data: [],
        totalField: 'total',
        dataField: 'rows',
        method: 'get',
        url: undefined,
        ajax: undefined,
        cache: true,
        contentType: 'application/json',
        dataType: 'json',
        ajaxOptions: {},
        queryParams: function (params) {
            return params;
        },
        queryParamsType: 'limit',       //undefined
        responseHandler: function (res) {
            return res;
        },
        pagination: false,
        onlyInfoPagination: false,
        paginationLoop: true,
        sidePagination: 'client',       //client or server
        totalRows: 0,                   //server side need to set
        pageNumber: 1,
        pageSize: 10,
        pageList: [10, 25, 50, 100],
        paginationHAlign: 'right',      //right, left
        paginationVAlign: 'bottom',     //bottom, top, both
        paginationDetailHAlign: 'left', //right, left
        paginationPreText: '‹',
        paginationNextText: '›',
        search: false,
        searchOnEnterKey: false,
        strictSearch: false,
        searchAlign: 'right',
        selectItemName: 'btSelectItem',
        showHeader: true,
        ...
}
```

13.10.3　实现收藏和删除图片的功能

下面我们来实现收藏图片和删除图片的功能。后端接口实现代码如下：

```
@Modifying
@Transactional
@Query("update #{#entityName} a set a.isFavorite=1,a.gmtModified=now()
where a.id=:id")
fun addFavorite(@Param("id") id: Long)
```

```
@Modifying
@Transactional
@Query("update #{#entityName} a set a.isDeleted=1 where a.id=:id")
fun delete(@Param("id") id: Long)
```

我们用 isFavorite=1 表示该图片是被收藏的，isDeleted=1 表示该图片被删除。需要注意的是 JPA 中 update、delete 操作需要在对应的函数上添加@Modifying 和@Transactional 注解。

控制层的 HTTP 接口代码如下：

```
@RequestMapping(value = "addFavorite", method = arrayOf(RequestMethod.POST))
@ResponseBody
fun addFavorite(@RequestParam(value = "id") id: Long): Boolean {
    imageRepository.addFavorite(id)
    return true
}

@RequestMapping(value = "delete", method = arrayOf(RequestMethod.POST))
@ResponseBody
fun delete(@RequestParam(value = "id") id: Long): Boolean {
    imageRepository.delete(id)
    return true
}
```

前端 JS 代码如下：

```
function addFavorite(id) {
    $.ajax({
        url: 'addFavorite',
        data: {id: id},
        dataType: 'json',
        type: 'post',
        success: function (resp) {
            //alert(JSON.stringify(resp))
            new PNotify({
                title: '收藏操作',
                styling: 'bootstrap3',
                text: JSON.stringify(resp),
                type: 'success',
                delay: 500,
            });
        },
        error: function (msg) {
            //alert(JSON.stringify(msg))
            new PNotify({
                title: '收藏操作',
                styling: 'bootstrap3',
                text: JSON.stringify(msg),
                type: 'error',
                delay: 500,
            });
        }
    })
}

function deleteById(id) {
    $.ajax({
        url: 'delete',
        data: {id: id},
```

```
        dataType: 'json',
        type: 'post',
        success: function (resp) {
            //alert(JSON.stringify(resp))
            $('#sotu favorite table').bootstrapTable('refresh')
            $('#sotu_table').bootstrapTable('refresh')
            new PNotify({
                title: '删除操作',
                styling: 'bootstrap3',
                text: JSON.stringify(resp),
                type: 'info',
                delay: 500,
            });
        },
        error: function (msg) {
            //alert(JSON.stringify(msg))
            new PNotify({
                title: '删除操作',
                styling: 'bootstrap3',
                text: JSON.stringify(msg),
                type: 'error',
                delay: 500,
            });
        }
    })
}
```

对应表格中的前端按钮组件代码在 sotu_table.js 中，关键代码片段如下：

```
{
    title: ' 操作',
    field: 'id',
    align: 'center',
    width: '5%',
    formatter: function (value, row, index) {
        var html = ""
        html += "<div onclick='addFavorite(" + value + ")' name= 'addFavorite'
       id='addFavorite" + value + "' class='btn btn-default'>收藏</div><p>"
        html += "<div onclick='deleteById(" + value + ")' name='delete'
       id='delete" + value + "' class='btn btn-default'>删除</div>"
        return html
    }
}
```

然后在 sotu_table.js 中，实现单击图片自动触发下载图片到本地的功能。代码如下：

```
{
    title: '美图',
    field: 'url',
    align: 'center',
    valign: 'middle',
    formatter: function (value, row, index) {
        return "![](" + value + ")"
    }
}
```

其中，downloadImage()函数实现代码如下：

```
function downloadImage(src) {
```

```
    var $a = $("<a></a>").attr("href", src).attr("download", "sotu.png");
    $a[0].click();
}
```

13.10.4 搜索关键字管理

本节我们开发管理爬虫爬取的关键字的功能。

1. 数据库实体类

首先，新建实体类 SearchKeyWord 如下：

```
package com.easy.kotlin.picturecrawler.entity

import java.util.*
import javax.persistence.*


@Entity
@Table(indexes = arrayOf(Index(name = "idx_key_word", columnList = "keyWord",
unique = true)))
class SearchKeyWord {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    var id: Long = -1
    @Column(name = "keyWord", length = 50, nullable = false, unique = true)
    var keyWord: String = ""
    @Column(nullable = true)
    var totalImage: Int? = 0
    var gmtCreated: Date = Date()
    var gmtModified: Date = Date()
    var isDeleted: Int = 0  //1 Yes, 0 No
    var deletedDate: Date = Date()
}
```

其中，keyWord 是搜索关键字，有唯一性约束，同时我们给它建立了索引。

2. dao层接口

接着来实现插入数据的 dao 层接口。

```
    @Modifying
    @Transactional
    @Query(value = "INSERT INTO 'search key word' ('deleted date', 'gmt created',
    'gmt modified', 'is deleted', 'key word') VALUES (now(), now(), now(),
    '0', :keyWord) ON
DUPLICATE KEY UPDATE 'gmt_modified' = now()", nativeQuery = true)
    fun saveOnNoDuplicateKey(@Param("keyWord") keyWord: String): Int
```

其中，ON DUPLICATE KEY UPDATE 这句代码表示当遇到重复的键值时，执行更新 gmt_modified=now()的操作。这里 nativeQuery=true，表示使用的是原生 SQL 查询。

3. 系统启动初始化动作

我们在应用启动类 PictureCrawlerApplication 中添加初始化动作：

```
package com.easy.kotlin.picturecrawler

import com.easy.kotlin.picturecrawler.dao.SearchKeyWordRepository
import com.easy.kotlin.picturecrawler.entity.SearchKeyWord
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.CommandLineRunner
import org.springframework.boot.SpringApplication
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.core.Ordered
import org.springframework.core.annotation.Order
import org.springframework.scheduling.annotation.EnableScheduling
import org.springframework.stereotype.Component
import java.io.File


@SpringBootApplication
@EnableScheduling
class PictureCrawlerApplication

fun main(args: Array<String>) {
    SpringApplication.run(PictureCrawlerApplication::class.java, *args)
}

@Component
@Order(value = Ordered.LOWEST_PRECEDENCE)
class initSearchKeyWordRunner : CommandLineRunner {
    @Autowired lateinit var searchKeyWordRepository: SearchKeyWordRepository

    override fun run(vararg args: String) {
        var keyWords = File("搜索关键词列表.data").readLines()
        keyWords.forEach {
            val SearchKeyWord = SearchKeyWord()
            SearchKeyWord.keyWord = it
            searchKeyWordRepository.saveOnNoDuplicateKey(it)
        }
    }
}
```

Spring Boot 应用程序在启动后会去遍历 CommandLineRunner 接口的实例并运行它们的 run 方法。使用@Order 注解来指定 CommandLineRunner 实例的运行顺序。

4. 搜索查询接口

查询所有关键字记录接口如下：

```
@Query("SELECT a from #{#entityName} a where a.isDeleted=0 order by a.id desc")
override fun findAll(pageable: Pageable): Page<SearchKeyWord>
```

模糊搜索关键字接口如下：

```
@Query("SELECT a from #{#entityName} a where a.isDeleted=0 and a.keyWord like
%:searchText% order by a.id desc")
fun search(@Param("searchText") searchText: String, pageable: Pageable):
Page<SearchKeyWord>
```

5. 模糊搜索HTTP接口的实现

与搜索图片分类的逻辑类似，模糊搜索关键字的接口如下：

```
    @RequestMapping(value = "searchKeyWordJson", method = arrayOf(Request-
    Method.GET))
    @ResponseBody
    fun sotuSearchJson(@RequestParam(value = "page", defaultValue = "0")
page: Int, @RequestParam(value = "size", defaultValue = "10") size: Int,
@RequestParam(value = "searchText", defaultValue = "") searchText: String):
Page<SearchKeyWord> {
        return getPageResult(page, size, searchText)
    }

    private fun getPageResult(page: Int, size: Int, searchText: String):
Page<SearchKeyWord> {
        val sort = Sort(Sort.Direction.DESC, "id")
        //注意：PageRequest.of(page,size,sort) page 默认是从 0 开始
        val pageable = PageRequest.of(page, size, sort)
        if (searchText == "") {
            return searchKeyWordRepository.findAll(pageable)
        } else {
            return searchKeyWordRepository.search(searchText, pageable)
        }
    }
```

6. 前端列表页面代码

search_keyword_view.ftl 模板页面代码如下

```
<#include 'common/head.ftl'>
<#include 'common/nav.ftl'>
<form id="add_key_word_form">
    <div class="col-lg-3">
        <div class="input-group">
            <input name="keyWord"
                   id="add_key_word_form_keyWord"
                   type="text"
                   class="form-control"
                   placeholder="输入爬虫抓取关键字">
            <span class="input-group-btn">
                        <button id="add_key_word_form_save_button"
                            class="btn btn-default"
                            type="button">
                             保存
                        </button>
            </span>
        </div><!-- /input-group -->
    </div><!-- /.col-lg-3 -->
</form>
<table id="search_keyword_table"></table>
<#include 'common/foot.ftl'>
<script src="search_keyword_table.js"></script>
```

search_keyword_table.js 代码如下：

```
$(function () {
    $.extend($.fn.bootstrapTable.defaults, $.fn.bootstrapTable.locales['zh-CN'])
    var searchText = $('.search').find('input').val()

    var columns = []

    columns.push(
```

```
        {
            title: 'ID',
            field: 'id',
            align: 'center',
            valign: 'middle',
            width: '10%',
            formatter: function (value, row, index) {
                return value
            }
        },
        {
            title: '关键字',
            field: 'keyWord',
            align: 'center',
            valign: 'middle',
            formatter: function (value, row, index) {
                var html = "<a href='sotu_view?keyWord=" + value + "' target=
                '_blank'>" + value + "</a>"
                return html
            }
        },
        {
            title: '图片总数',
            field: 'totalImage',
            align: 'center',
            valign: 'middle',
            formatter: function (value, row, index) {
                var html = "<a href='sotu_view?keyWord=" + row.keyWord + "'
                 target='_blank'>" + row.totalImage + "</a>"
                return html
            }
        })

    $('#search_keyword_table').bootstrapTable({
        url: 'searchKeyWordJson',
        sidePagination: "server",
        queryParamsType: 'page,size',
        contentType: "application/x-www-form-urlencoded",
        method: 'get',
        striped: false,    //是否显示行间隔色
        cache: false,  //是否使用缓存，默认为true，所以一般情况下需要设置一下这个属性（*）
        pagination: true,  //是否显示分页（*）
        paginationLoop: true,
        paginationHAlign: 'right',          //right, left
        paginationVAlign: 'bottom',         //bottom, top, both
        paginationDetailHAlign: 'left',     //right, left
        paginationPreText: ' 上一页',
        paginationNextText: '下一页',
        search: true,
        searchText: searchText,
        searchTimeOut: 500,
        searchAlign: 'right',
        searchOnEnterKey: false,
        trimOnSearch: true,
        sortable: true,                     //是否启用排序
        sortOrder: "desc",                  //排序方式
        sortName: "id",
        pageNumber: 1,                      //初始化加载第一页，默认第一页
        pageSize: 10,                       //每页的记录行数（*）
```

```
        pageList: [8, 16, 32, 64, 128], //可选的每页数据
        totalField: 'totalElements',     //所有记录 count
       dataField: 'content',             //后端 json 对应的表格 List 数据的 key
        columns: columns,
        queryParams: function (params) {
            return {
                size: params.pageSize,
                page: params.pageNumber - 1,
                sortName: params.sortName,
                sortOrder: params.sortOrder,
                searchText: params.searchText
            }
        },
        classes: 'table table-responsive full-width',
    })
})
```

7. 添加爬取关键字

添加爬取关键字 HTTP 接口代码如下

```
@RequestMapping(value = "save_keyword", method = arrayOf(RequestMethod.GET,
RequestMethod.POST))
@ResponseBody
fun save(@RequestParam(value = "keyWord")keyWord:String): String {
    if(keyWord==""){
        return "0"
    }else{
        searchKeyWordRepository.saveOnNoDuplicateKey(keyWord)
        return "1"
    }
}
```

前端输入框表单代码如下：

```
<form id="add_key_word_form">
    <div class="col-lg-3">
        <div class="input-group">
            <input name="keyWord"
                   id="add_key_word_form_keyWord"
                   type="text"
                   class="form-control"
                   placeholder="输入爬虫抓取关键字">
            <span class="input-group-btn">
                        <button id="add_key_word_form_save_button"
                            class="btn btn-default"
                            type="button">
                             保存
                        </button>
            </span>
        </div><!-- /input-group -->
    </div><!-- /.col-lg-3 -->
</form>
```

对应的 JS 代码如下：

```
$('#add_key_word_form_save_button').on('click', function () {
    var keyWord = $('#add_key_word_form_keyWord').val()
    $.ajax({
        url: 'save_keyword',
```

```
        type: 'get',
        data: {keyWord: keyWord},
        success: function (response) {
            if (response == "1") {
                alert("保存成功")
                $('#search_keyword_table').bootstrapTable('refresh')
            } else {
                alert("数据不能为空")
            }
        },
        error: function (error) {
            alert(JSON.stringify(error))
        }
    })
})
```

其中，$('#search_keyword_table').bootstrapTable('refresh')是保存成功后，刷新表格后的内容。

8．更新该关键字的图片总数

下面来实现统计一个关键字对应的图片总数列表的功能，效果如图 13-13 所示。

搜图　美图列表　精选收藏　搜索关键字　系统定时任务　Kotlin　关于

输入爬虫抓取关键字　保存　　搜索

ID	关键字	图片总数
37	颜真卿书法	226
36	王羲之书法	357
35	动物世界	465
34	太阳系	548
33	地球	878
32	太空	648
31	星空	1033
30	宇宙	1007
29	沙漠	552
28	蓝天白云	488

显示第 131 到第 140 条记录，总共 167 条记录 每页显示 10 条记录　　上一页 1 ... 13 14 15 16 17 下一页

图 13-13　搜索关键字对应的图片总数列表

更新 search_key_word 表 total_image 字段的 SQL 逻辑代码如下：

```
@Modifying
@Transactional
@Query("update search_key_word a set a.total_image = (select count(*) from
image i where i.is_deleted=0 and i.category like concat('%',a.key_word,'%'))",
nativeQuery = true)
fun batchUpdateTotalImage()
```

表示该对应关键字所包含的图片总数。然后开始执行这个定时任务，代码如下：

```
package com.easy.kotlin.picturecrawler.job

import com.easy.kotlin.picturecrawler.dao.SearchKeyWordRepository
import kotlinx.coroutines.experimental.CommonPool
```

```
import kotlinx.coroutines.experimental.launch
import kotlinx.coroutines.experimental.runBlocking
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.scheduling.annotation.Scheduled
import org.springframework.stereotype.Component
import java.util.*

@Component
class BatchUpdateJob {

    @Autowired lateinit var searchKeyWordRepository: SearchKeyWordRepository

    @Scheduled(cron = "0 */5 * * * ?")
    fun job() {
        println("开始执行定时任务 batchUpdateTotalImage:  ${Date()}")
        searchKeyWordRepository.batchUpdateTotalImage()
    }
}
```

13.10.5 使用协程实现异步爬虫任务

上面我们的定时任务都是同步的。当我们想用 HTTP 接口去触发任务执行的时候，可能并不想一直等待，这个时候可以使用异步的方式。这里我们使用 Kotlin 提供的轻量级线程——协程来实现。在常用的并发模型中，多进程、多线程、分布式是最普遍的，不过近些年来逐渐有一些语言以 first-class 或者 library 的形式提供对基于协程并发模型的支持。其中比较典型的有 Scheme、Lua、Python、Perl、Go 等以 first-class 方式提供对协程的支持。同样，Kotlin 也支持协程。（关于协程的更多介绍，可参考《Kotlin 极简教程》第 9 章轻量级线程：协程）

在 build.gradle 中添加 kotlinx-coroutines-core 依赖：

```
compile group: 'org.jetbrains.kotlinx', name: 'kotlinx-coroutines-core',
version: '0.19.2'
```

然后把定时任务代码修改如下：

```
@Component
class BatchUpdateJob {

    @Autowired lateinit var searchKeyWordRepository: SearchKeyWordRepository

    @Scheduled(cron = "0 */5 * * * ?")
    fun job() {
        doBatchUpdate()
    }

    fun doBatchUpdate() {
        launch(CommonPool) {
            println("开始执行定时任务 batchUpdateTotalImage:  ${Date()}")
            searchKeyWordRepository.batchUpdateTotalImage()
        }
    }
}
```

同样，爬虫抓取图片的任务也可以进行如下改写：

```
fun doCrawJob(){
    val list = searchKeyWordRepository.findAll()
    for (i in 1..1000) {
        list.forEach {
            launch(CommonPool) {
                saveImage(it.keyWord, i)
            }
        }
    }
}
```

其中，launch()函数会以非阻塞（non-blocking）当前线程的方式，启动一个新的协程后台任务，并返回一个 Job 类型的对象作为当前协程的引用。我们把真正要执行的代码逻辑放到 launch(CommonPool){ //... }中，这样就可以手动启动任务进行异步执行了。

13.10.6　图片存入数据库并在前端展现

数据库实体类如下：

```
package com.easy.kotlin.picturecrawler.entity

import java.util.*
import javax.persistence.*

@Entity
@Table(indexes = arrayOf(
        Index(name = "idx_url", unique = true, columnList = "url"),
        Index(name = "idx_category", unique = false, columnList = "category")))
class Image {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    var id: Long = -1
    @Version
    var version: Int = 0

    @Column(length = 255, unique = true, nullable = false)
    var category: String = ""
    var isFavorite: Int = 0

    @Column(length = 255, unique = true, nullable = false)
    var url: String = ""

    var gmtCreated: Date = Date()
    var gmtModified: Date = Date()
    var isDeleted: Int = 0  //1 Yes, 0 No
    var deletedDate: Date = Date()

    @Lob
    var imageBlob: ByteArray = byteArrayOf()
    /* 0-Baidu  1-Gank */
    var sourceType: Int = 0

    override fun toString(): String {
        return "Image(id=$id, version=$version, category='$category', isFavorite=
        $isFavorite, url='$url', gmtCreated=$gmtCreated, gmtModified=
        $gmtModified, isDeleted=$isDeleted, deletedDate=$deletedDate)"
    }
}
```

其中，@Lob var imageBlob: ByteArray = byteArrayOf()这个字段存储图片的 Base64 内容。当然，我们在生产环境中通常不会这样存储图片等文件，而是单独放在文件服务器上，我们在数据库里只存图片 url 即可。这里为了方便演示，就直接存进数据库里了。把图片比特流数组存入数据库代码如下：

```
val image = Image()
image.category = "干货集中营福利"
image.url = url
image.sourceType = 1
image.imageBlob = getByteArray(url)
logger.info("Image = ${Image}")
imageRepository.save(Image)
```

其中的 getByteArray(url) 函数代码如下：

```
private fun getByteArray(url: String): ByteArray {
    val urlObj = URL(url)
    return urlObj.readBytes()
}
```

前端 HTML 展示图片的代码如下：

```
{
        title: '图片',
        field: 'imageBlob',
        align: 'center',
        valign: 'middle',
        formatter: function (value, row, index) {
            //var html = "<img onclick=downloadImage('" + value + "')
            width='100%' src='" + value + "'>"
            var html = '<img onclick="downBase64Image(this.src)" width=
             "100%" src="data:image/jpg;base64,' + value + '"/>'
            return html
        }
}
```

单击下载的 JS 代码如下：

```
function downloadImage(src) {
    var $a = $("<a></a>").attr("href", src).attr("download", "sotu.png");
    $a[0].click();
}

function downBase64Image(url) {
    var blob = base64Img2Blob(url);
    url = window.URL.createObjectURL(blob);
    var $a = $("<a></a>").attr("href", url).attr("download", "sotu.png");
    $a[0].click();
}

function base64Img2Blob(code) {
    var parts = code.split(';base64,');
    var contentType = parts[0].split(':')[1];
    var raw = window.atob(parts[1]);
    var rawLength = raw.length;
    var uInt8Array = new Uint8Array(rawLength);

    for (var i = 0; i < rawLength; ++i) {
        uInt8Array[i] = raw.charCodeAt(i);
```

```
    }

    return new Blob([uInt8Array], {type: contentType});
}
```

本节完整的项目源码地址是 https://github.com/EasySpringBoot/picture-crawler

13.11　本章小结

在 Spring Framework 5.0 中已经添加了对 Kotlin 的支持。使用 Kotlin 集成 Spring Boot 开发非常流畅自然，几乎不需要任何迁移成本。所以，Kotlin 在未来的 Java 服务端领域也必将受到越来越多的程序员的关注。

第 14 章　使用 Kotlin 进行 Android 开发

本章将带领大家快速入门如何使用 Kotlin 进行 Android 应用程序的开发。

在 Realm Report(2017-Q4，https://realm.io/realm-report/2017-q4)中，在 2016 年 9 月至 2017 年 9 月 Android 端的开发：Java 从 95%降低到了 85%，而 Kotlin 从 5%涨到了 15%，如图 14-1 所示。

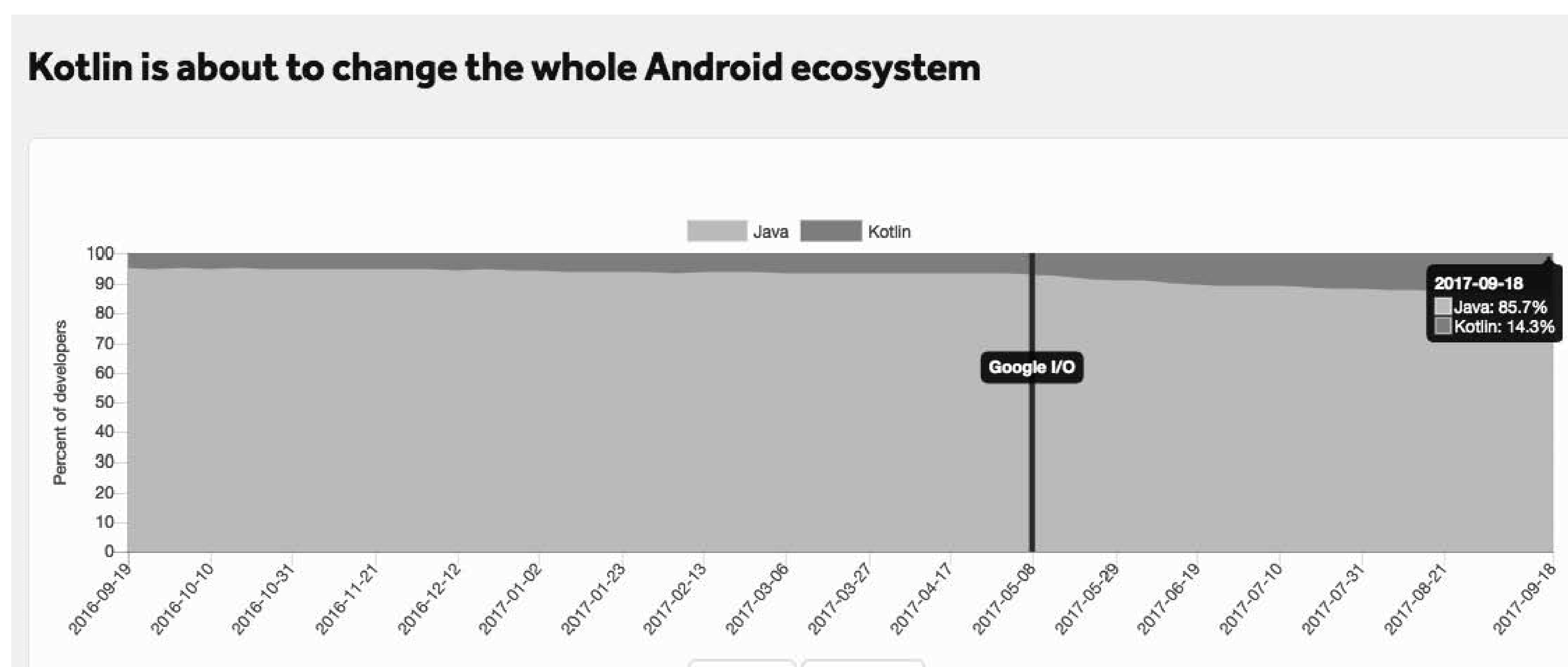


图 14-1　Kotlin 在 Android 领域 2016 年 9 月至 2017 年 9 月的占比情况

Kotlin 在 Android 领域 2018 年的占比预测如图 14-2 所示。

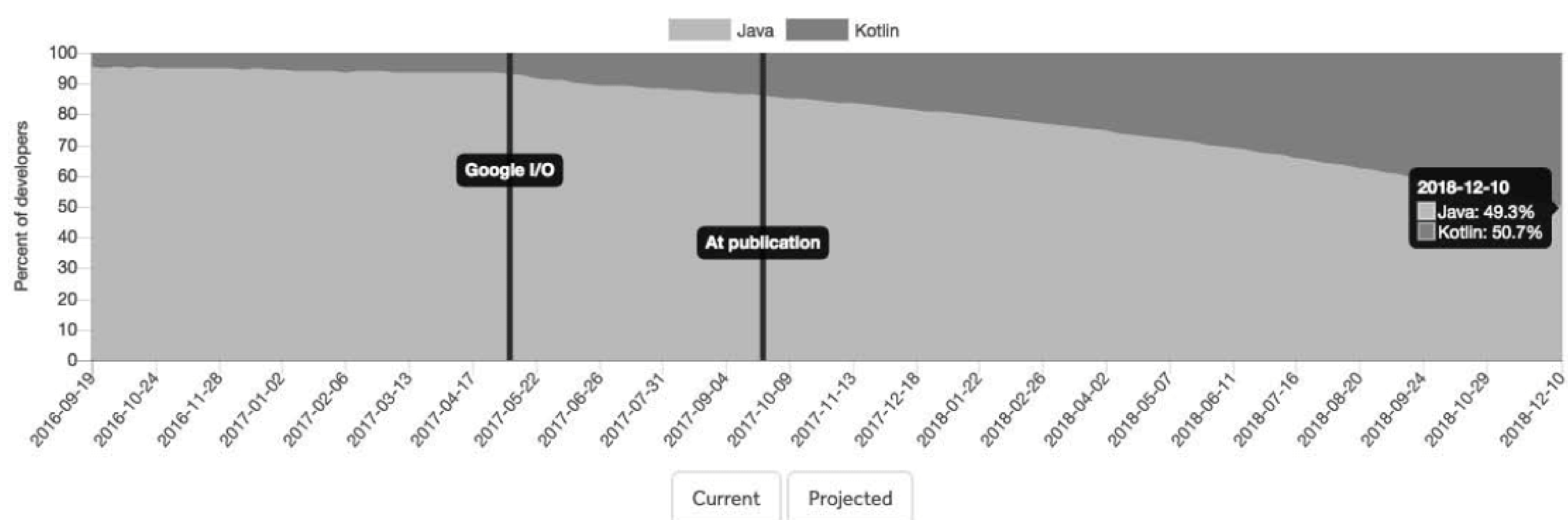


图 14-2　Kotlin 在 Android 领域 2018 年的占比预测

从这个趋势来看，加上最新 Android Studio 3.0 的发布（内置 Kotlin 开发 Android 项目的支持），Kotlin 将会很快颠覆 Java 在 Android 领域的地位。

14.1　快速开发 Hello World

本节我们从一个简单的 Kotlin 版本的 Hello World Android 应用程序开始。

14.1.1　准备工作

首先准备好开发工具。Android 开发还是建议用 Google 官方支持的 IDE：Android Studio。

1. Android Studio 3.0 简介

Google 在 2017 年 10 月 26 日发布了 Android 8.1 首个开发者预览版的同时还正式发布了 Android Studio 3.0，为其 IDE 引入了一系列新功能。

Android Studio 3.0 专注于加速 Android 应用开发，包含大量更新内容，其中最主要的功能之一就包括对 Kotlin 的支持。正如 Google 在 2017 年的 I/O 开发者大会上所宣布的那样，Kotlin 已被官方支持用于 Android 开发。Android Studio 3.0 是第一个支持 Kotlin 语言的里程碑式版本（在此之前，可以使用 Android Studio 的 Kotlin 插件）。

在 Android Studio 3.0 中提供了许多方便且实用的功能，如代码自动补全和语法高亮显示。另外，Android Studio 内置转换工具可以非常方便地把 Java 代码转换成 Kotlin 代码，如图 14-3 所示。

```
1   package com.easy.kotlin.myjavahelloworld.feature;
2
3   import android.support.v7.app.AppCompatActivity;
4   import android.os.Bundle;
5
6   public class MainActivity extends AppCompatActivity {
7
8       @Override
9       protected void onCreate(Bundle savedInstanceState) {
10          super.onCreate(savedInstanceState);
11          setContentView(R.layout.activity_main);
12      }
13  }
14
15
16
```

图 14-3　打开要转换的 Java 代码源文件

首先，打开要转换的 Java 代码源文件 MainActivity.java，如图 14-3 所示。

然后在菜单栏中依次选择 Code | Convert Java File to Kotlin File 命令，如图 14-4 所示。



图 14-4　依次选择 Code | Convert Java File to Kotlin File 命令

选择上面的命令后，将看到转换之后的 Kotlin 代码 MainActivity.kt，如图 14-5 所示。



图 14-5　转换之后的 Kotlin 代码

2．安装Android Studio 3.0

Android Studio 是 Android 的官方 IDE。Android Studio 3.0 的一个亮点就是内置了 Kotlin 的支持，可参看 Google 官方介绍：https://developer.android.google.cn/kotlin/index.html。

在 Google 2017 年的 I/O 开发者大会上，Kotlin 已成为 Google 钦定的 Android 官方开发语言。

使用 Android Studio 3.0，可以方便地把 Java 源代码自动转换成 Kotlin 代码，也可以直接创建 Kotlin 语言开发的 Android 项目,只需要在新建项目的时候勾选 Include Kotlin support 即可。

首先去官网 https://developer.android.google.cn/studio/install.html 下载安装包。笔者当前下载的安装包版本是 android-studio-ide-171.4408382-mac.dmg，下载完毕后单击 dmg 文件，如图 14-6 所示。



图 14-6　下载完毕单击 dmg 文件

然后将其复制到应用程序中即可。

14.1.2　创建基于 Kotlin 的 Android 项目

首先新建项目。如果尚未打开项目，请在 Welcome to Android Studio 窗口中选择 Start a new Android Studio project 选项，如图 14-7 所示。

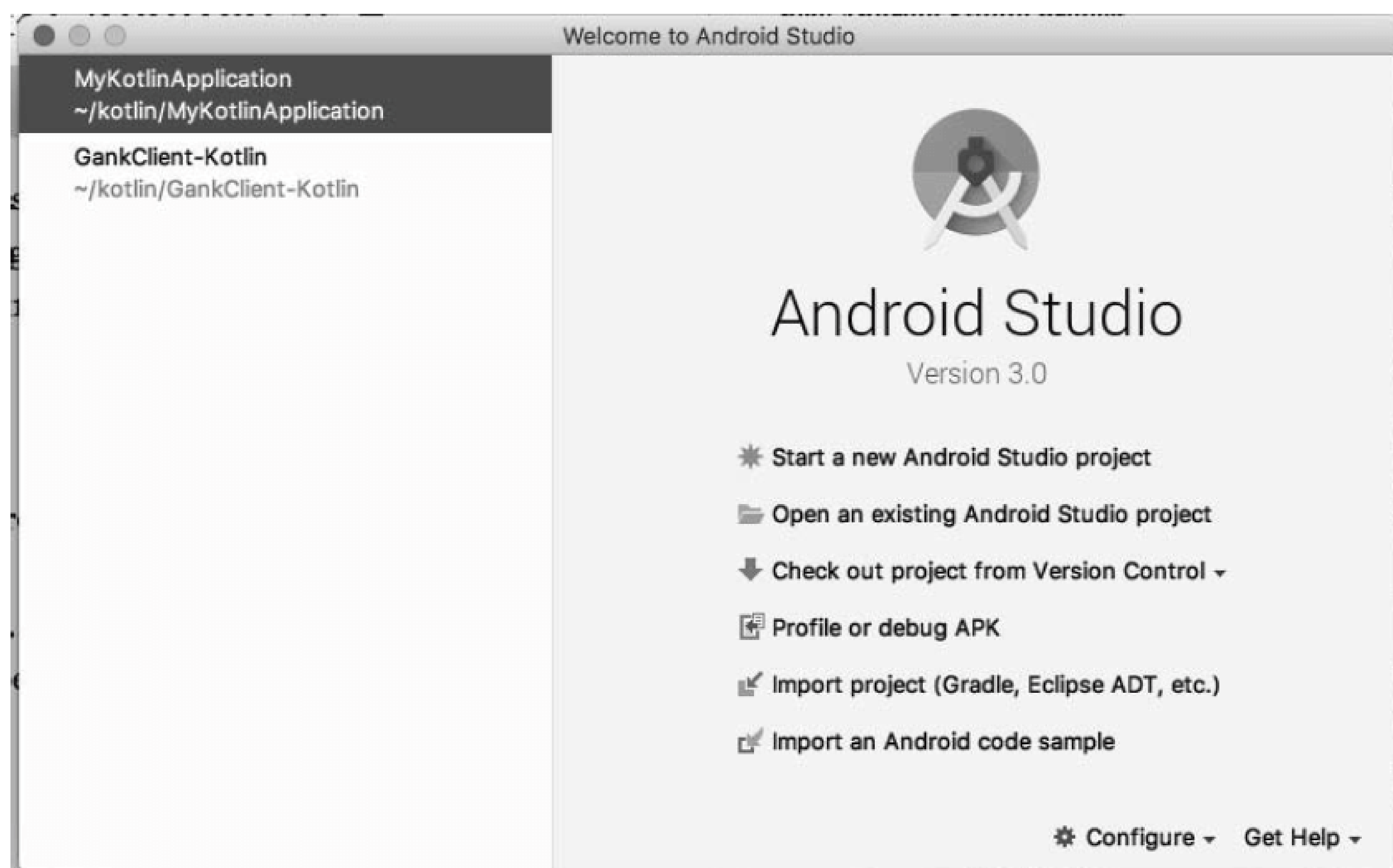


图 14-7　选择 Start a new Android Studio project 选项

如果已经打开项目，依次选择 File | New | New Project 命令，如图 14-8 所示。



图 14-8　选择 File | New | New Project 命令

此时弹出 Create Android Project 对话框。在创建 Android 项目对话框中配置应用基本信息，注意勾选 Kotlin 支持选项，单击 Next 按钮，如图 14-9 所示。

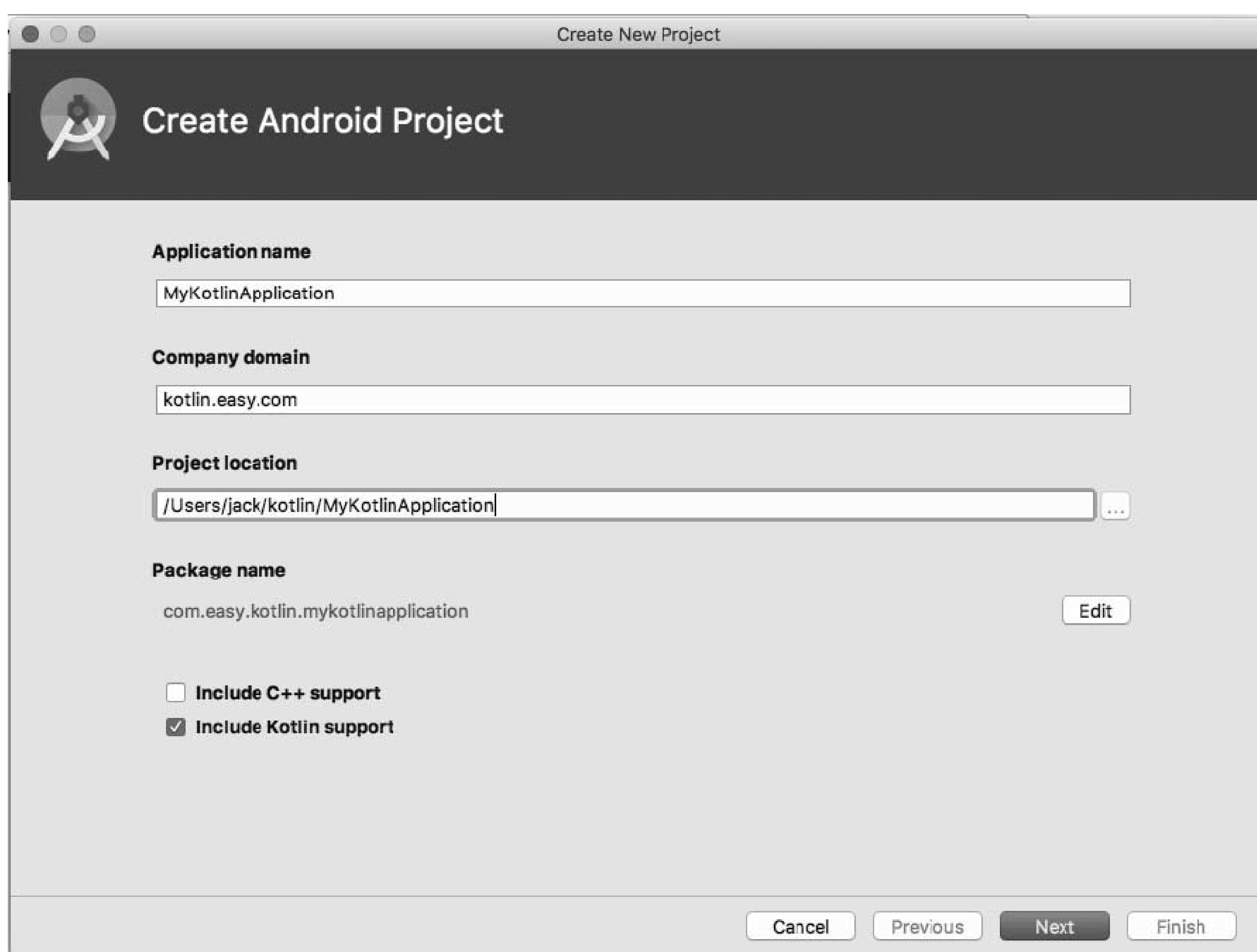


图 14-9　创建 Android 项目

此时进入 Create Android Project 界面，在其中配置应用运行 SDK 及环境信息，如图 14-10 所示。

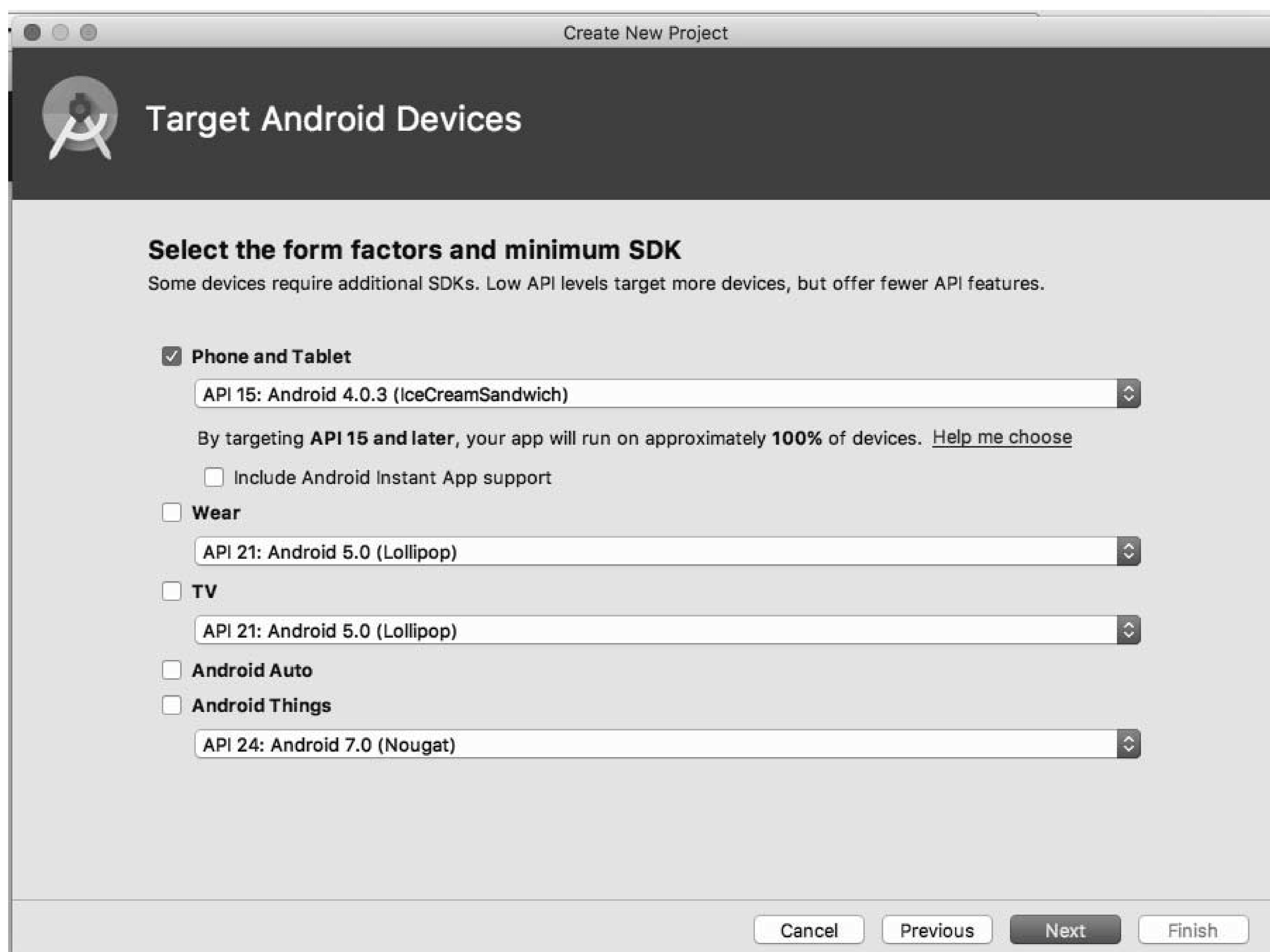


图 14-10　配置应用运行 SDK 及环境信息

然后勾选 Phone and Tablet 复选框，然后选择 API 15：Android 4.0.3，单击 Next 按钮进入添加 Activity 界面，如图 14-11 所示。

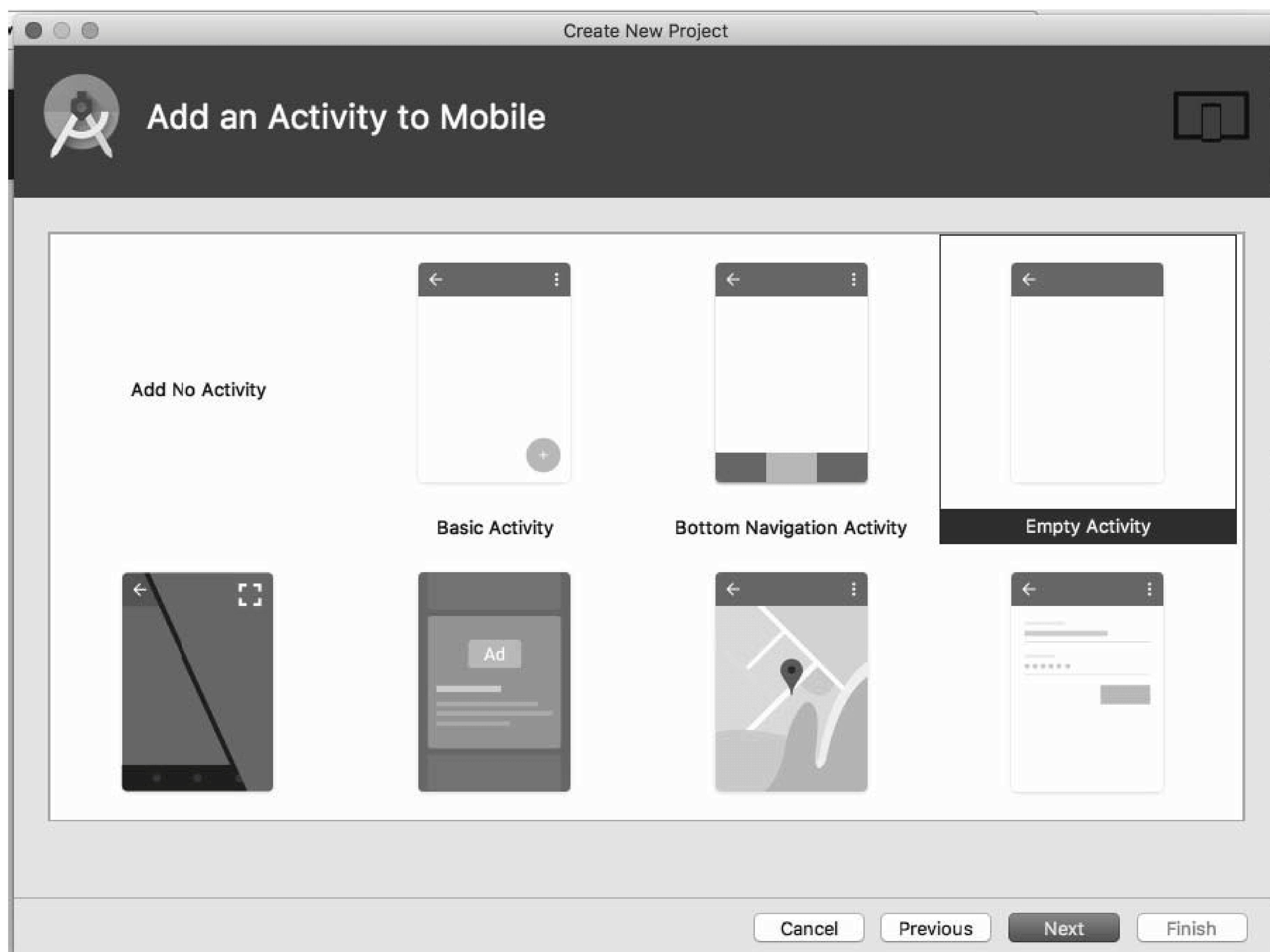


图 14-11　添加 Activity 界面

这里选择 Empty Activity，然后单击 Next 按钮进入配置 Activity 界面，如图 14-12 所示。

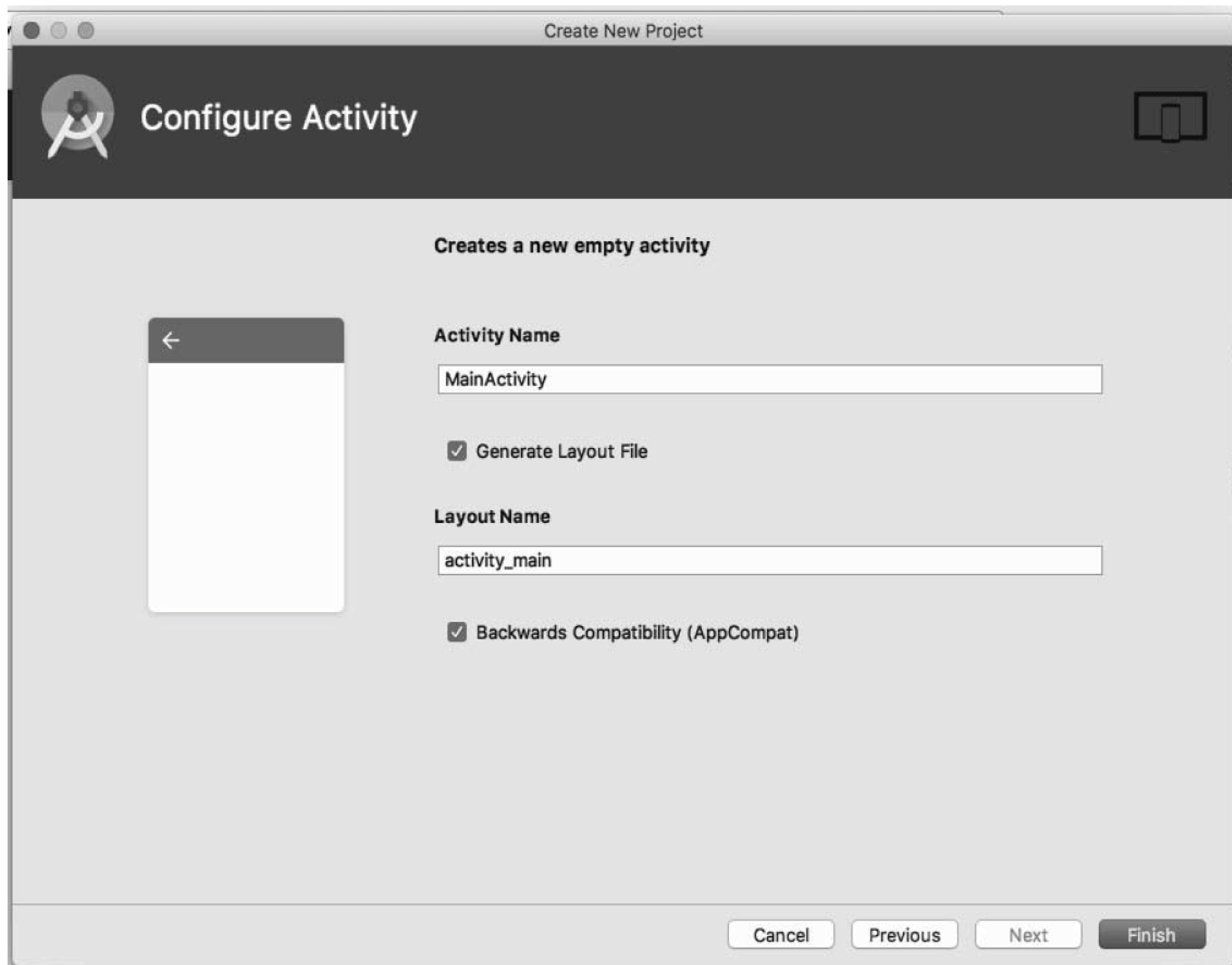


图 14-12　配置 Activity 界面

配置好 Activity Name 与 Layout Name 之后，单击 Finish 按钮。之后我们将得到一个 Kotlin 版本的 Hello World 的 Android 应用程序。工程目录如图 14-13 所示。

14.1.3　工程目录文件说明

其中，在顶层的 Gradle 配置文件 build.gradle 中添加了 kotlin-gradle-plugin 插件的依赖：

```
buildscript {
    ext.kotlin version = '1.1.51'
    ...
    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.0'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
}
...
```

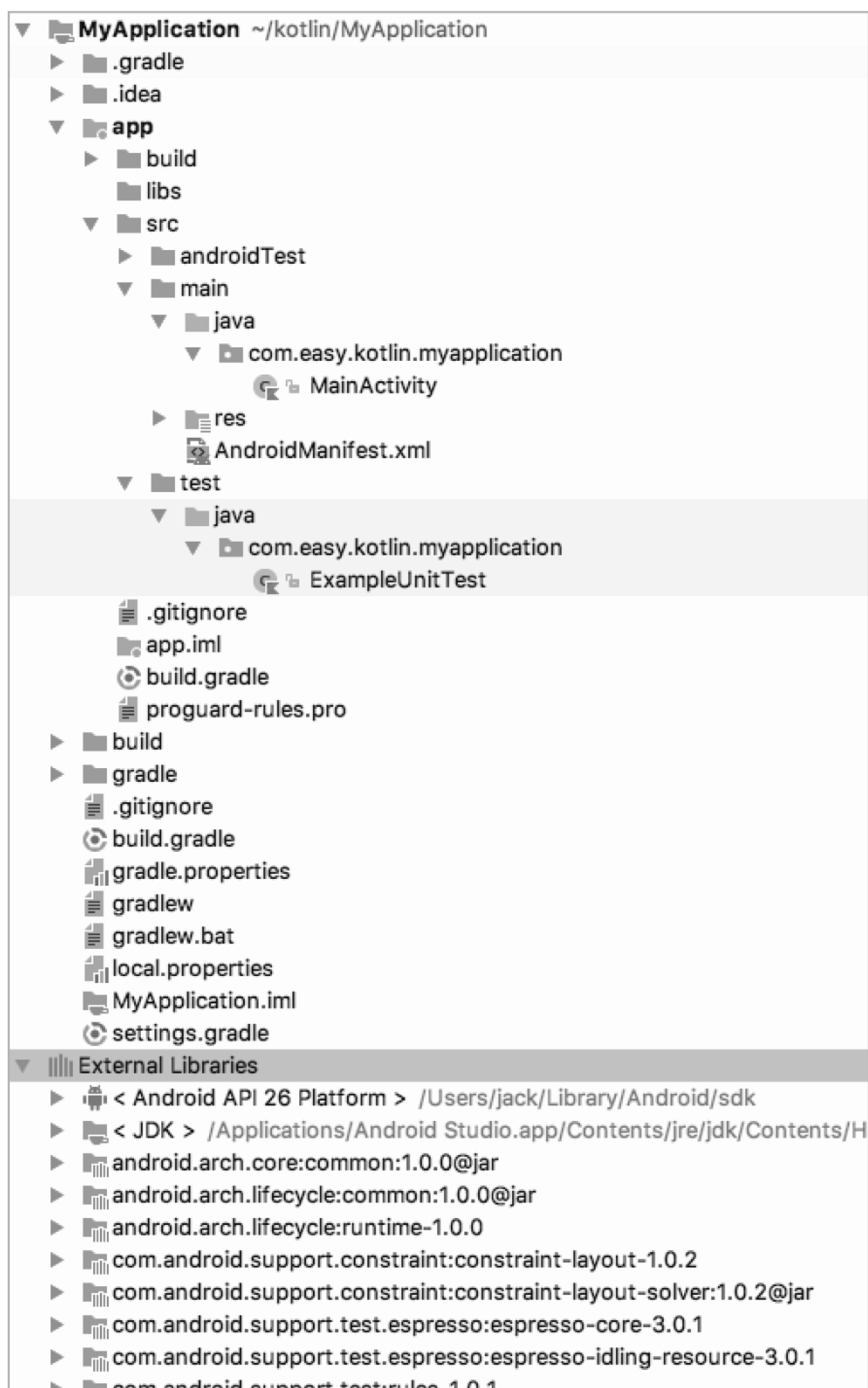


图 14-13　工程目录

app 目录下的 build.gradle 配置文件内容如下：

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

dependencies {
```

```
    ...
    implementation"org.jetbrains.kotlin:kotlin-stdlib-jre7:$kotlin_version"
    ...
}
```

其中的 apply plugin:'kotlin-android-extensions'表示使用 Kotlin Android Extensions 插件。这个插件是 Kotlin 专门针对 Android 扩展的插件，实现了与 Data-Binding、Dagger 等框架的功能。

布局文件 activity_main.xml 内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.easy.kotlin.myapplication.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

MainActivity.kt 代码如下：

```
package com.easy.kotlin.myapplication

import android.support.v7.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

AndroidManifest.xml 文件内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.easy.kotlin.myapplication">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
```

```
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

14.1.4　安装运行

单击功能菜单栏中的运行按钮，运行程序，如图 14-14 所示。

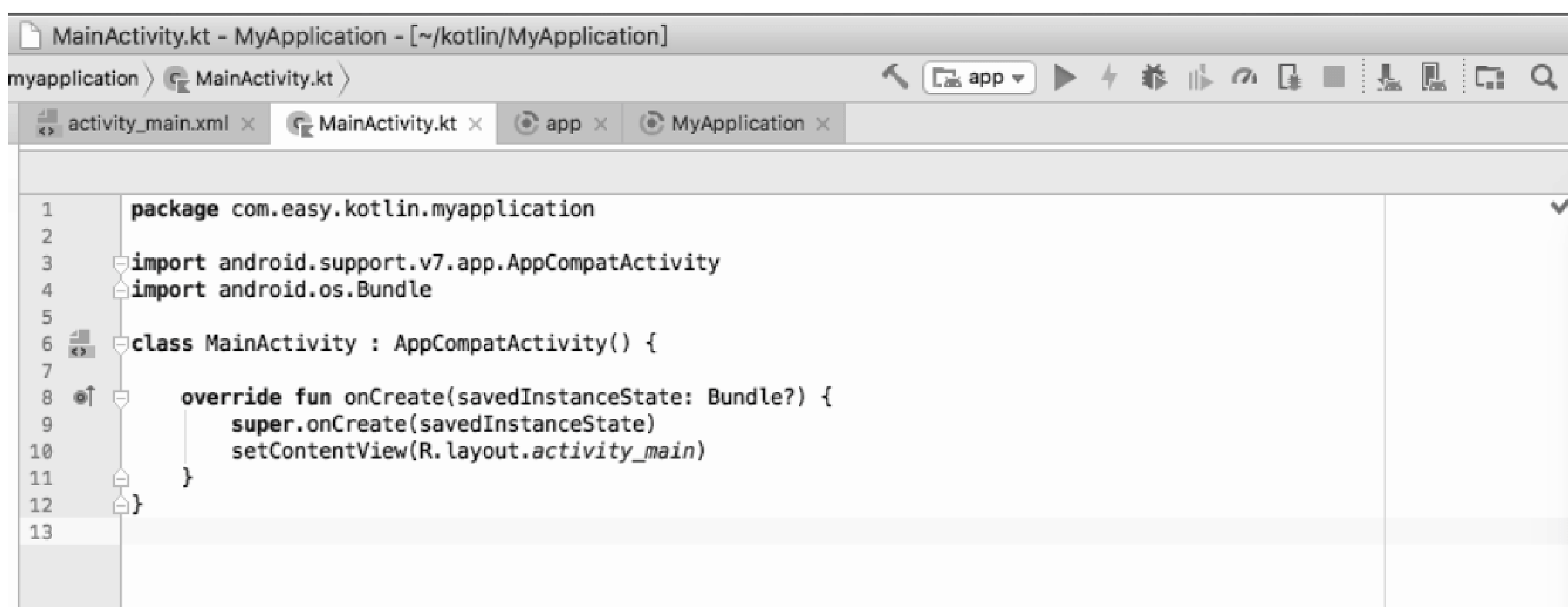


图 14-14　运行应用程序

此时会提示我们选择应用程序部署运行的目标设备，如图 14-15 所示。



图 14-15　选择部署设备

需要注意的是，手机要开启 USB 调试模式。单击 OK 按钮，Android Studio 会为我们完成打包、安装等事项。最终的运行效果如图 14-16 所示。

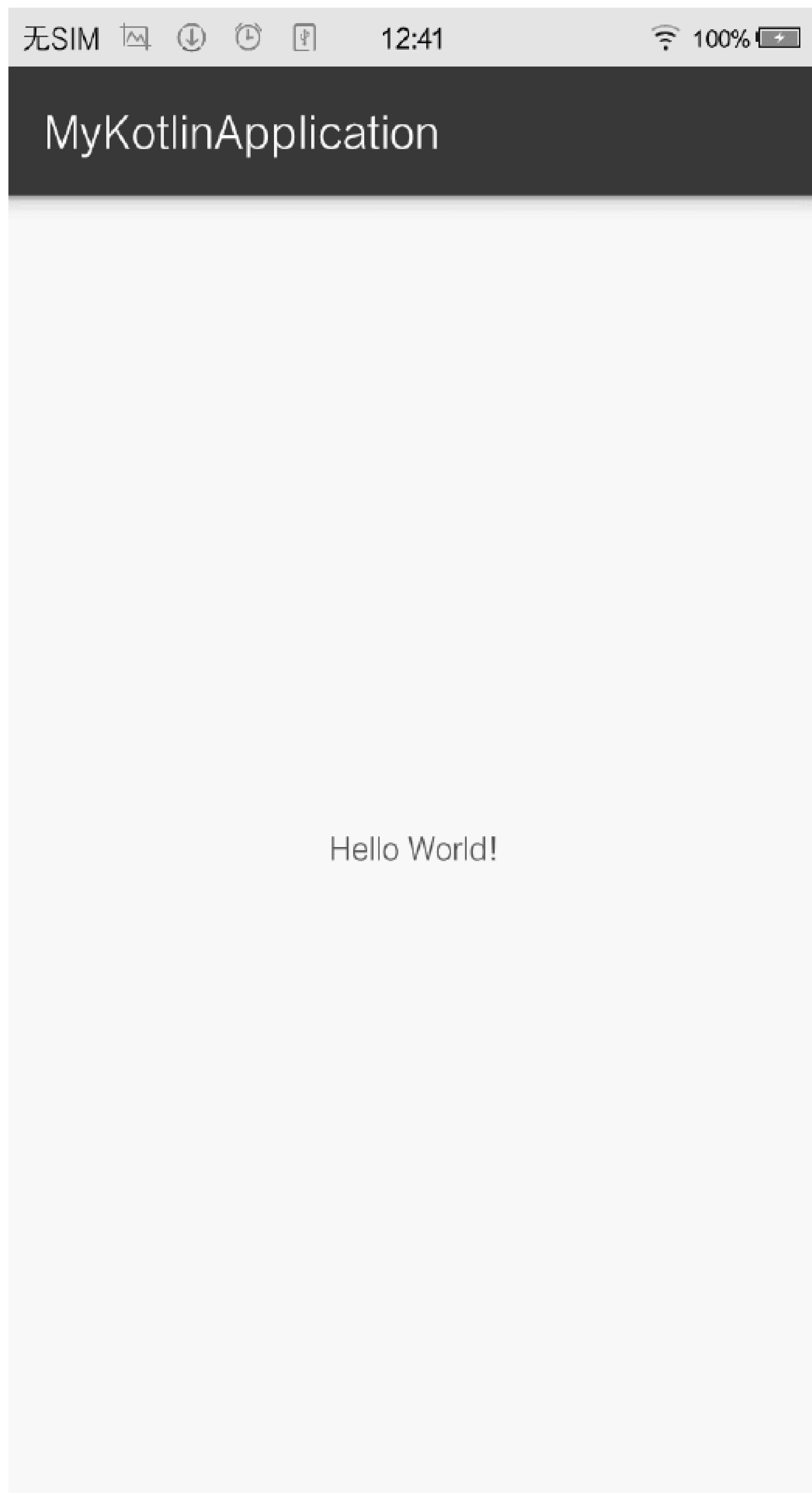


图 14-16 Hello World 运行效果

14.2 综合项目实战：开发一个电影指南应用程序

下面我们来开发一个电影指南 Android 应用程序，列出流行/最高评级的电影，显示预告片和评论。

14.2.1 创建 Kotlin Android 项目

首先创建一个 Kotlin Android 项目，然后单击 Next 按钮，如图 14-17 所示。

在 Target Android Devices 界面中选择目标设备后单击 Next 按钮，如图 14-18 所示。

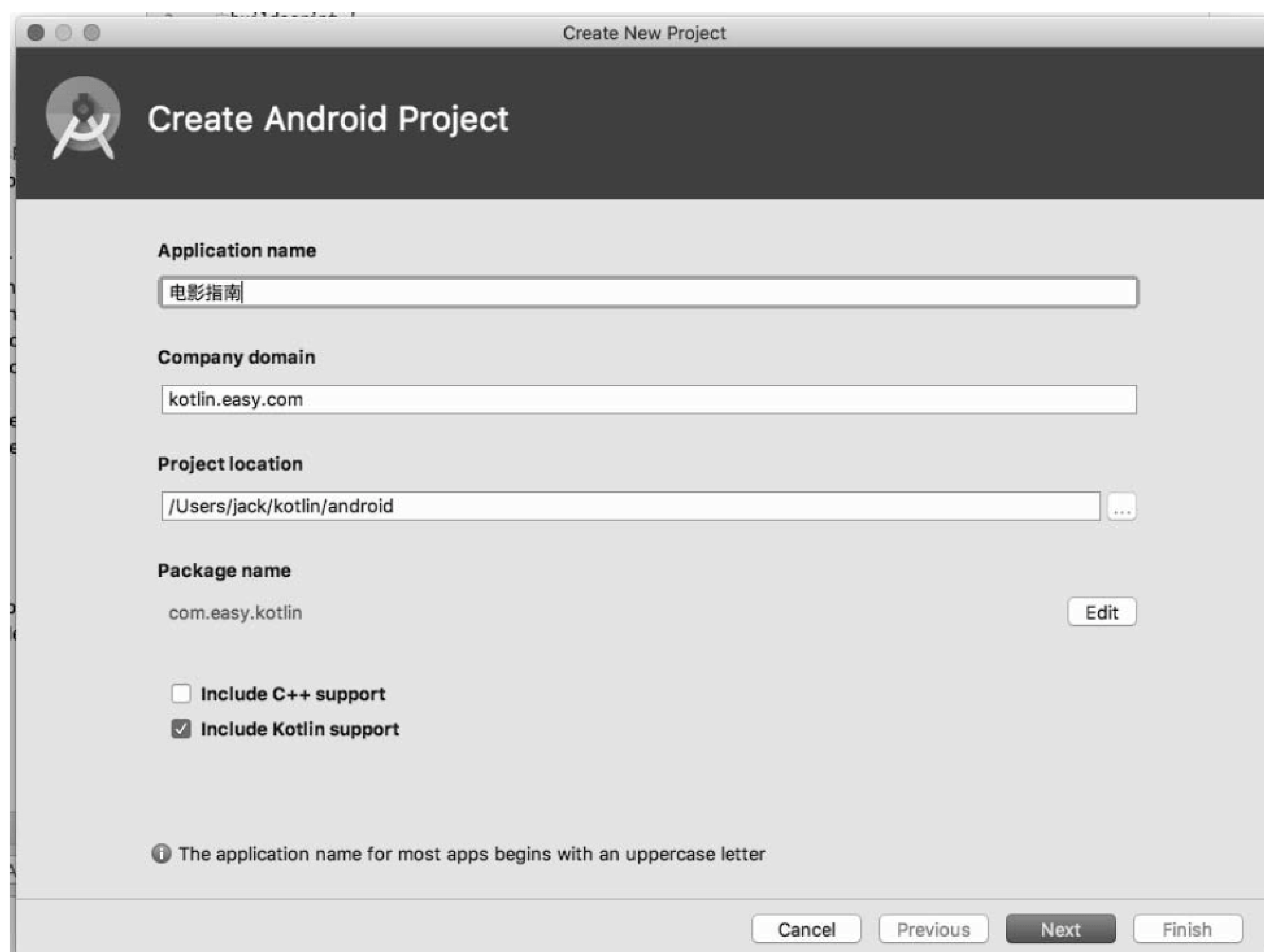


图 14-17　创建 Kotlin Android 项目

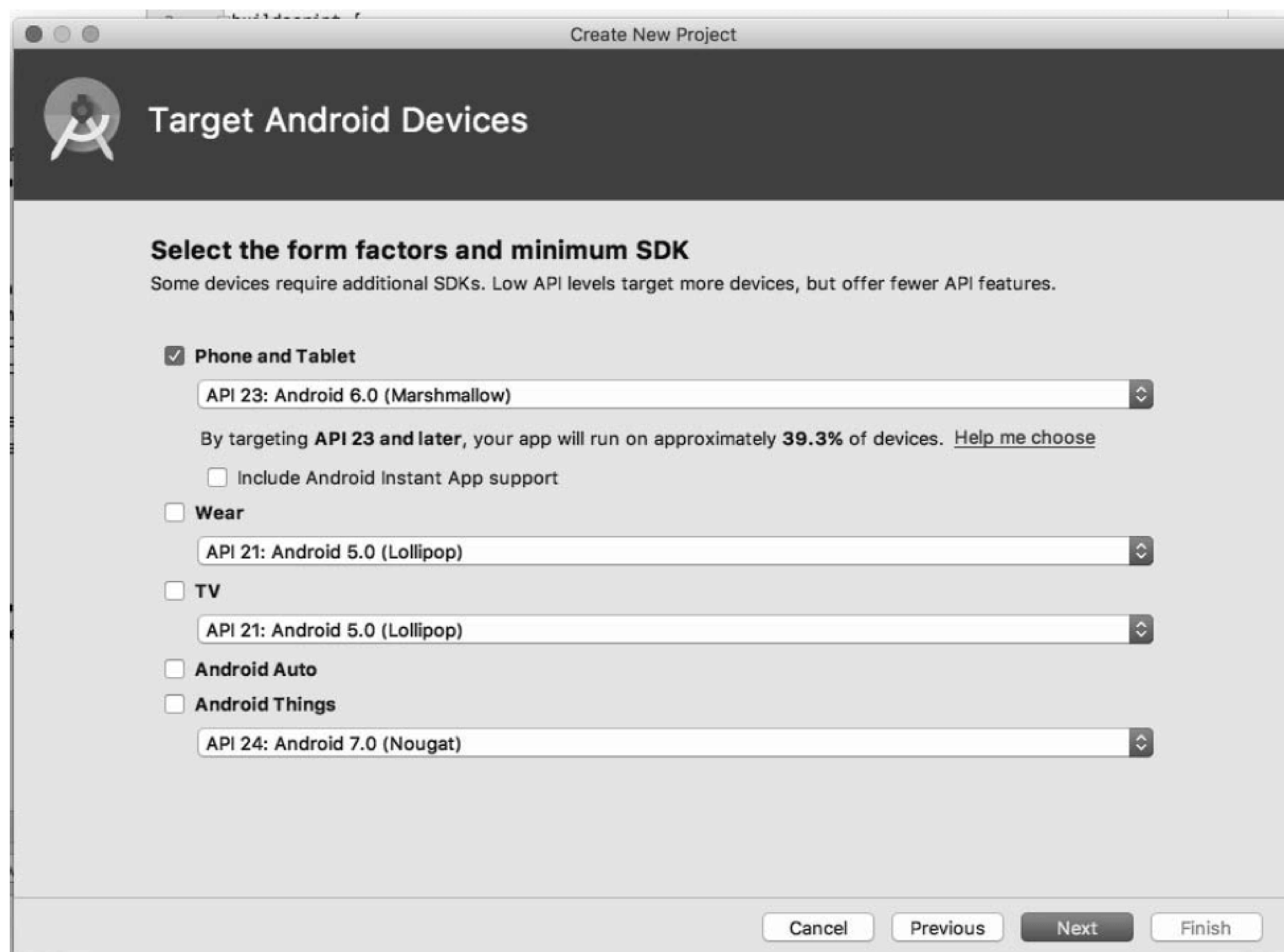


图 14-18　选择目标设备

然后在 Add an Activity to Mobile 界面中添加一个 Master/Detail Flow，单击 Next 按钮，如图 14-19 所示。

在 Configure Activity 界面中配置 Activity，单击 Finish 完成配置，如图 14-20 所示。

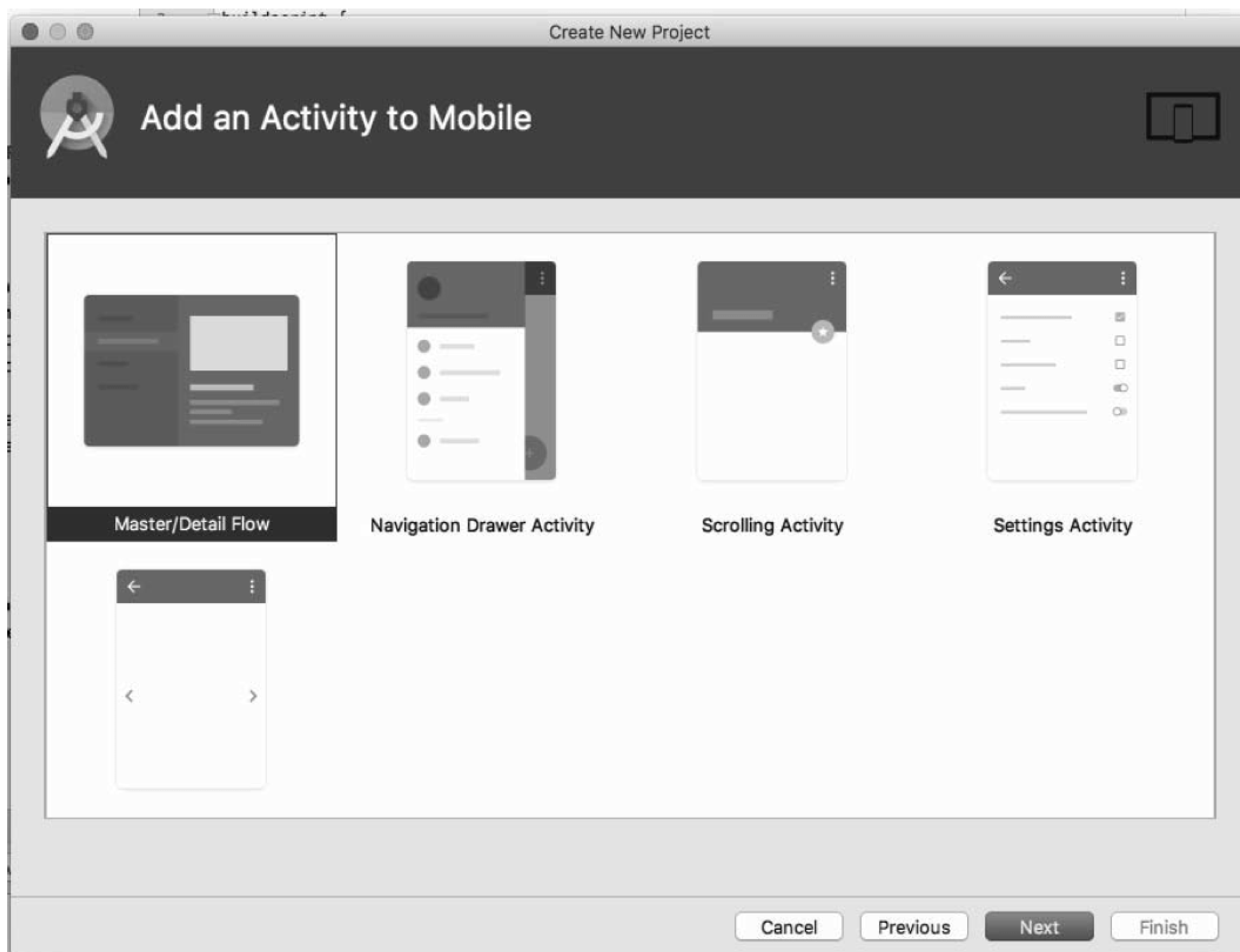


图 14-19　添加 Master/Detail Flow

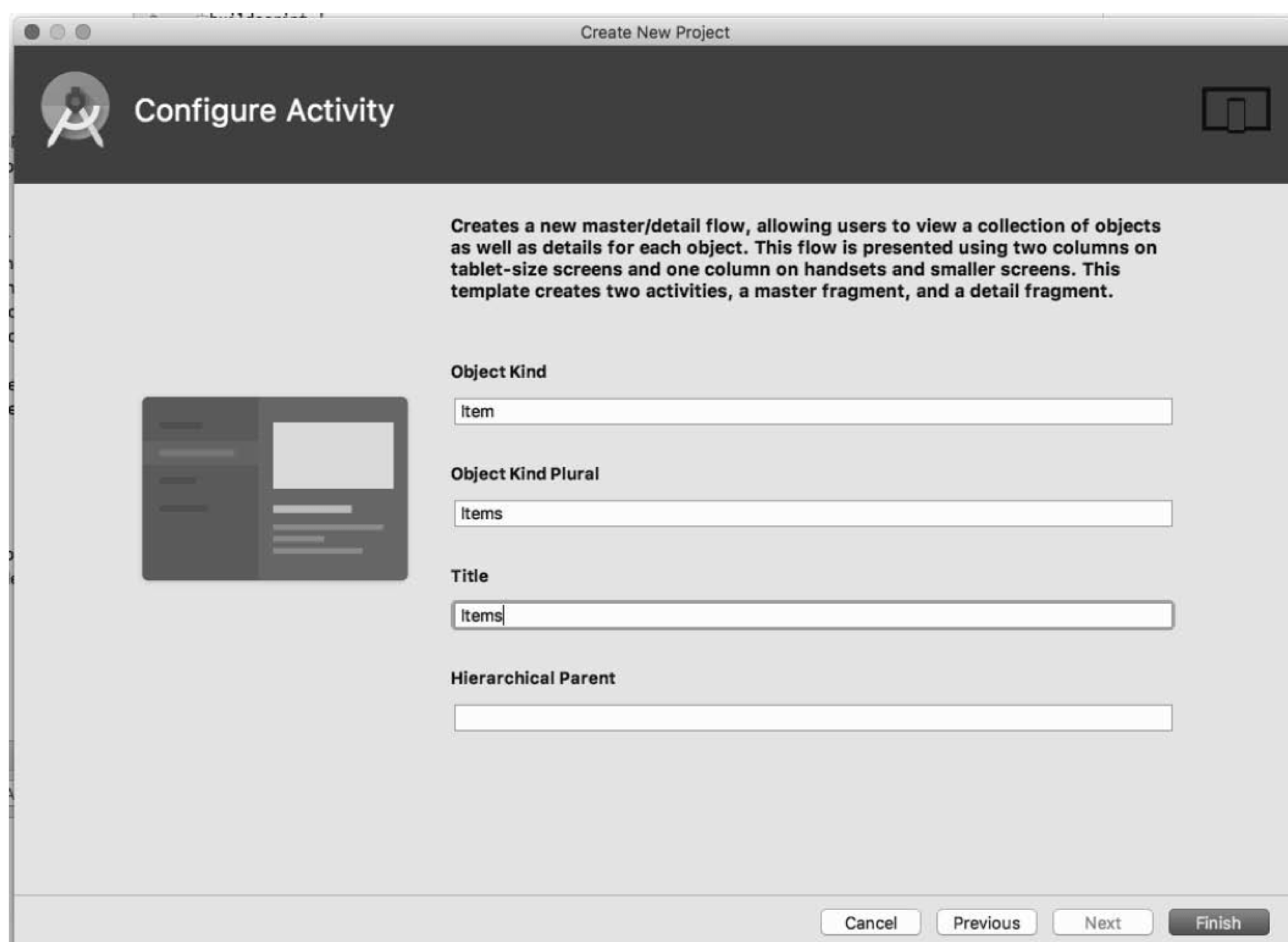


图 14-20　配置 Activity

最终生成的 Android 项目工程目录结构如图 14-21 所示。

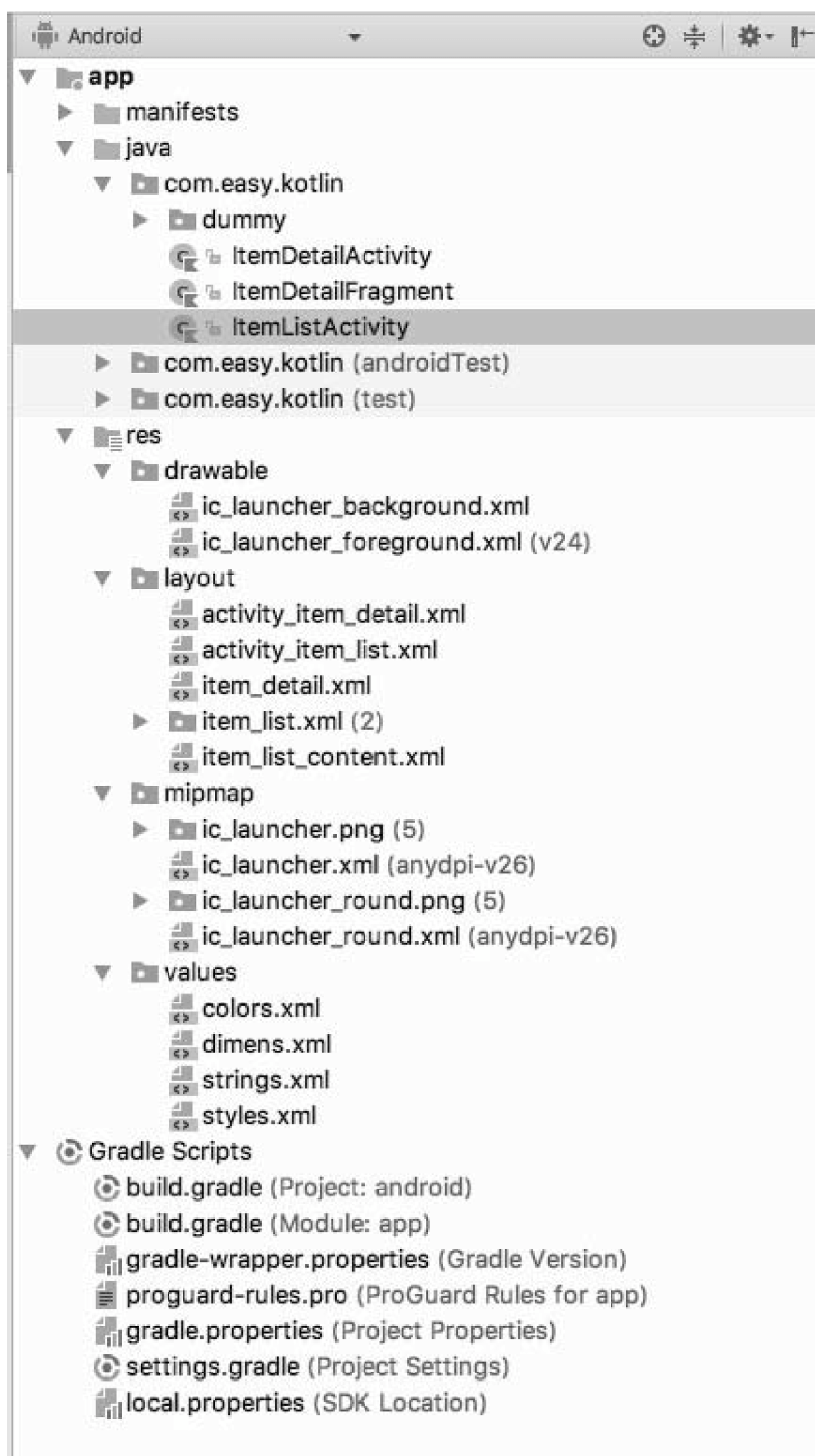
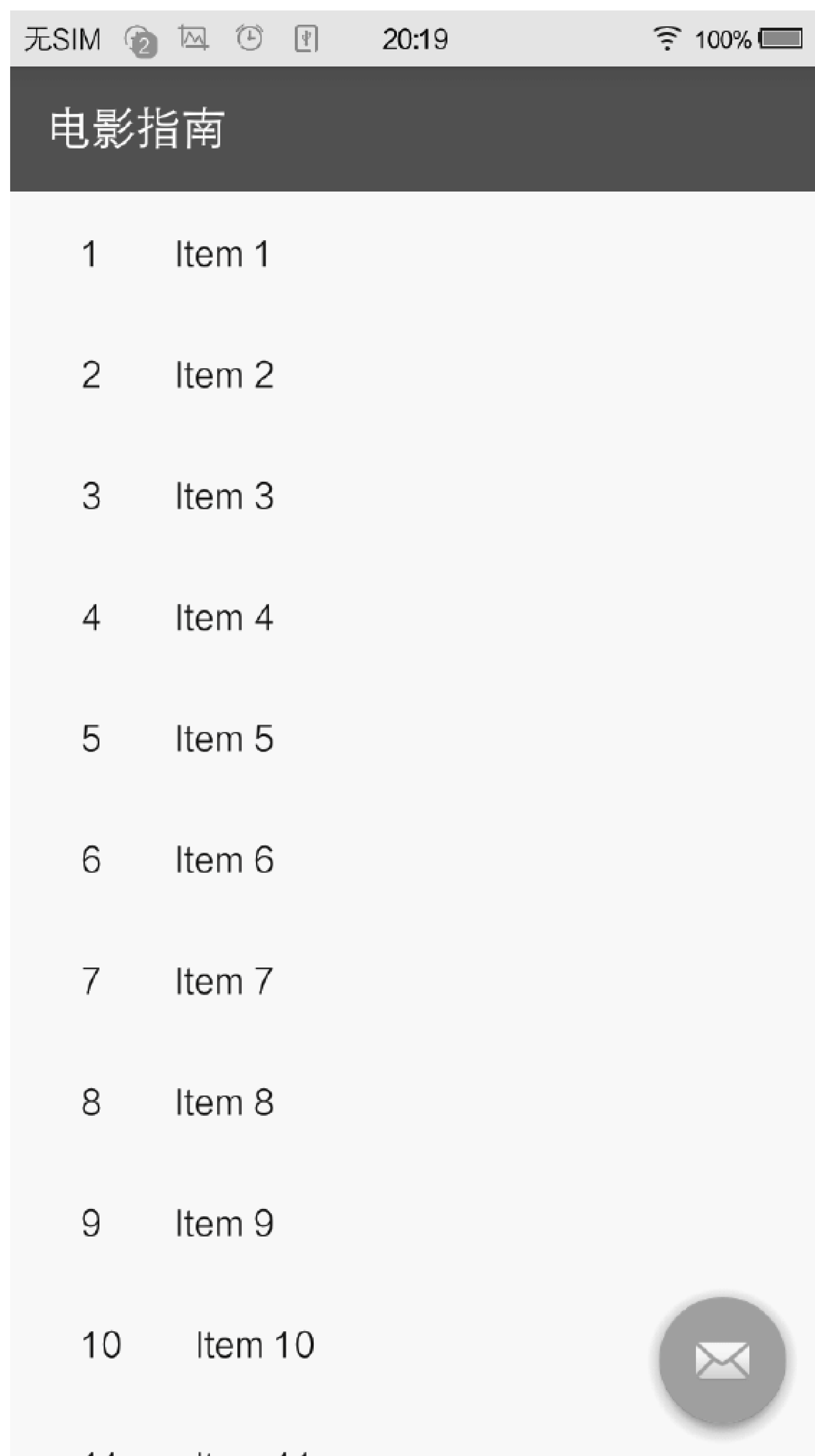


图 14-21　Android 项目工程目录结构

运行之后的列表页如图 14-22 所示。

选择 Item1 进入详情页，如图 14-23 所示。



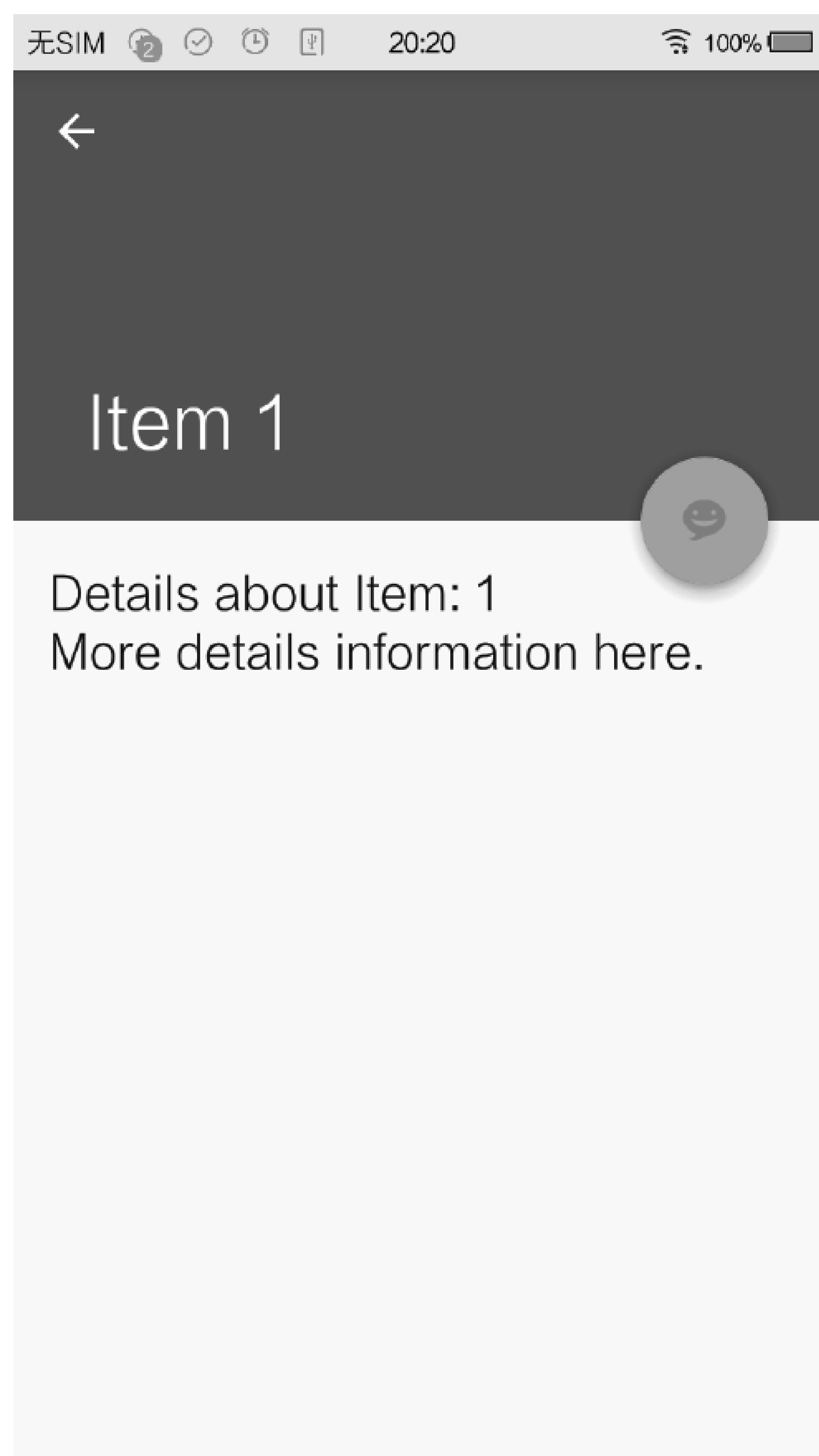


图 14-22　运行之后的列表页

图 14-23　进入 Item1 详情页

其中 AndroidManifest.xml 代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.easy.kotlin">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".ItemListActivity"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
```

```
        </activity>
        <activity
            android:name=".ItemDetailActivity"
            android:label="@string/title item detail"
            android:parentActivityName=".ItemListActivity"
            android:theme="@style/AppTheme.NoActionBar">
            <meta-data
                android:name="android.support.PARENT ACTIVITY"
                android:value="com.easy.kotlin.ItemListActivity" />
        </activity>
    </application>

</manifest>
```

其中，android.intent.action.MAIN 处的配置指定了应用程序的启动 Activity 为 .ItemListActivity，其中的点号“.”表示该类位于 package="com.easy.kotlin"路径下。

```
<activity
    android:name=".ItemListActivity"
    android:label="@string/ app name"
    android:theme="@style/AppTheme.NoActionBar">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

14.2.2　启动主类 ItemListActivity

下面来介绍应用程序的启动主类 ItemListActivity。ItemListActivity 的 Kotlin 代码如下：

```
package com.easy.kotlin

import android.content.Intent
import android.os.Bundle
import android.support.v7.app.AppCompatActivity
import android.support.v7.widget.RecyclerView
import android.support.design.widget.Snackbar
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView

import com.easy.kotlin.dummy.DummyContent
import kotlinx.android.synthetic.main.activity_item_list.*
import kotlinx.android.synthetic.main.item_list_content.view.*

import kotlinx.android.synthetic.main.item_list.*

class ItemListActivity : AppCompatActivity() {

    /**
     * Whether or not the activity is in two-pane mode, i.e. running on a tablet
     * device.
     */
    private var mTwoPane: Boolean = false
```

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity item list)

    setSupportActionBar(toolbar)
    toolbar.title = title

    fab.setOnClickListener { view ->
        Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
                .setAction("Action", null).show()
    }

    if (item_detail_container != null) {
        //The detail container view will be present only in the
        //large-screen layouts (res/values-w900dp).
        //If this view is present, then the
        //activity should be in two-pane mode.
        mTwoPane = true
    }
    setupRecyclerView(item_list) //设置 RecyclerView
}

private fun setupRecyclerView(recyclerView: RecyclerView) {
    recyclerView.adapter = SimpleItemRecyclerViewAdapter(this,DummyContent.
    ITEMS, mTwoPane)
}

class SimpleItemRecyclerViewAdapter(
    private val mParentActivity: ItemListActivity,
    private val mValues: List<DummyContent.DummyItem>,
    private val mTwoPane: Boolean) :
RecyclerView.Adapter<SimpleItemRecyclerViewAdapter.ViewHolder>() {

    private val mOnClickListener: View.OnClickListener
    init {
        mOnClickListener = View.OnClickListener { v ->
            val item = v.tag as DummyContent.DummyItem
            if (mTwoPane) {
                val fragment = ItemDetailFragment().apply {
                    arguments = Bundle()
                    arguments.putString(ItemDetailFragment.ARG ITEM ID,
                    item.id)
                }
                mParentActivity.supportFragmentManager
                        .beginTransaction()
                        .replace(R.id.item detail container, fragment)
                        .commit()
            } else {
                val intent = Intent(v.context, ItemDetailActivity::class.
                java).apply {
                    putExtra(ItemDetailFragment.ARG ITEM ID, item.id)
                }
                v.context.startActivity(intent)
            }
        }
    }
```

```
        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
        ViewHolder {
            //设置 LayoutInflater
            val view = LayoutInflater.from(parent.context)
                    .inflate(R.layout.item_list_content, parent, false)
            return ViewHolder(view)
        }

        override fun onBindViewHolder(holder: ViewHolder, position: Int) {
            val item = mValues[position]
            holder.mIdView.text = item.id
            holder.mContentView.text = item.content

            with(holder.itemView) {
                tag = item
                setOnClickListener(mOnClickListener)
            }
        }

        override fun getItemCount(): Int {
            return mValues.size
        }

        inner class ViewHolder(mView: View) : RecyclerView.ViewHolder(mView) {
            val mIdView: TextView = mView.id_text
            val mContentView: TextView = mView.content
        }
    }
}
```

布局文件 XML 代码中的 activity_item_list.xml 代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context="com.easy.kotlin.ItemListActivity">

    <android.support.design.widget.AppBarLayout
        android:id="@+id/app_bar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

    </android.support.design.widget.AppBarLayout>
```

```
    <FrameLayout
        android:id="@+id/frameLayout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior">

        <include layout="@layout/item_list" />
    </FrameLayout>

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        app:srcCompat="@android:drawable/ic_dialog_email" />

</android.support.design.widget.CoordinatorLayout>
```

对应的 UI 设计效果图如图 14-24 所示。

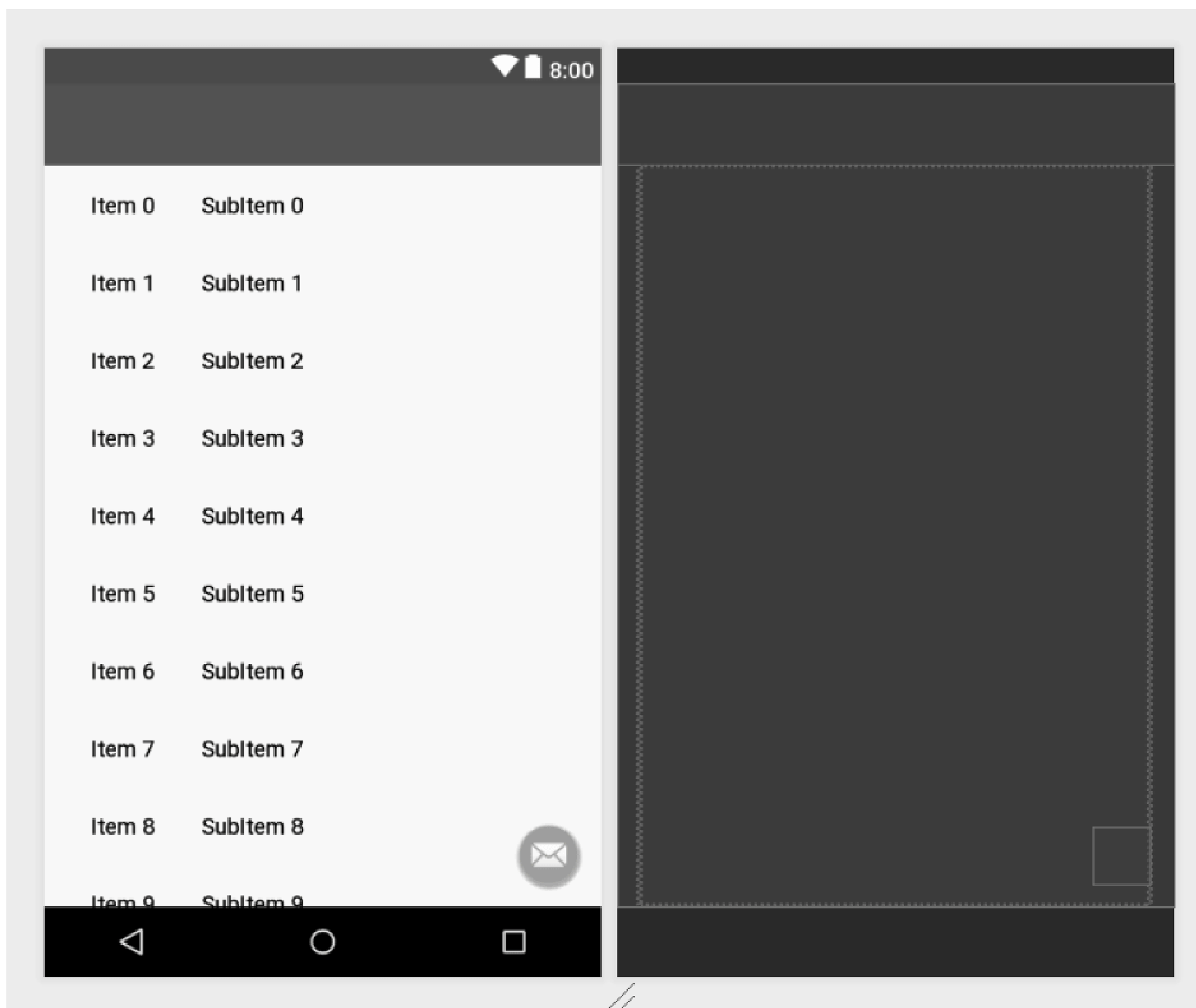


图 14-24　列表页的 UI 设计图

14.2.3　AppCompatActivity 类介绍

在使用 Android Studio 开发 Android 应用的时候，创建项目时，自动继承的是

AppCompatActivity。这样我们可以在自定义的 Activity 类中添加 android.support.v7.app.ActionBar（APIlevel7+）。例如 activity_item_list.xml 布局中的

```
<android.support.design.widget.AppBarLayout
    android:id="@+id/app_bar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:theme="@style/AppTheme.AppBarOverlay">

    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        app:popupTheme="@style/AppTheme.PopupOverlay" />

</android.support.design.widget.AppBarLayout>
```

Activity 中添加 Toolbar 的代码如下：

```
class ItemListActivity : AppCompatActivity() {

    /**
     * Whether or not the activity is in two-pane mode, i.e. running on a tablet
     * device.
     */
    private var mTwoPane: Boolean = false

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_item_list)

        setSupportActionBar(toolbar)
        toolbar.title = title

        fab.setOnClickListener { view ->
            Snackbar.make(view, "Replace with your own action", Snackbar.
            LENGTH_LONG).setAction("Action", null).show()
        }

        if (item_detail_container != null) {
            //The detail container view will be present only in the
            //large-screen layouts (res/values-w900dp).
            //If this view is present, then the
            //activity should be in two-pane mode.
            mTwoPane = true
        }

        setupRecyclerView(item_list)
    }
}
```

AppCompatActivity 背后继承的也是 Activity。Android 5.0 推出之后，提供了很多新功能，于是 support v7 也更新了，出现了 AppCompatActivity。AppCompatActivity 是用来替代 ActionBarActivity 的。AppCompatActivity 的类图继承层次如图 14-25 所示。

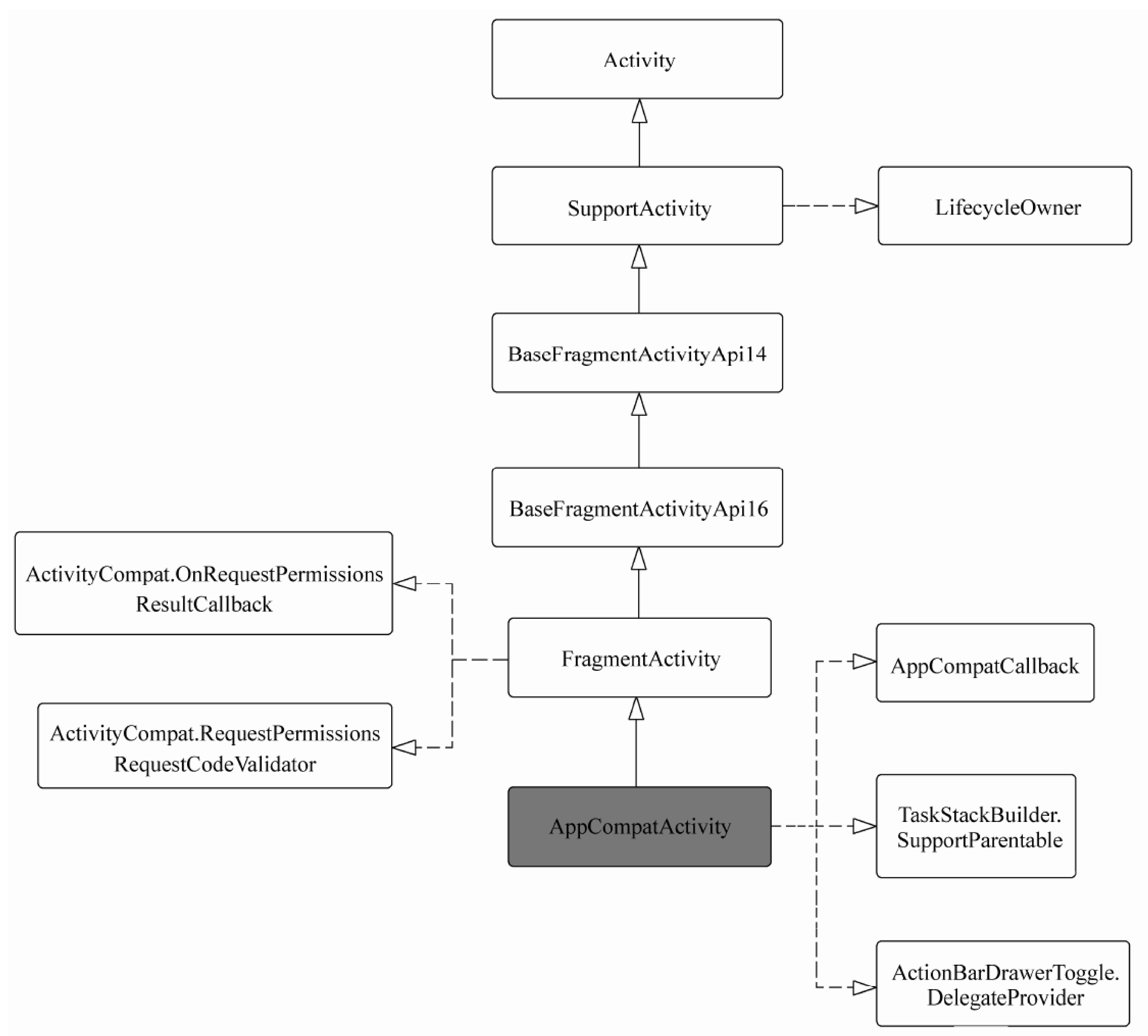


图 14-25　AppCompatActivity 的类图继承层次

14.2.4　Activity 生命周期

Activity 的生命周期示意图如图 14-26 所示（图来自官网）。

相信不少朋友已经看过这个流程图了，这里面简单说明一下。

（1）开始启动 Activity，系统会先调用 onCreate()方法，然后调用 onStart()方法，最后调用 onResume()方法，Activity 进入运行状态。

（2）当前 Activity 被其他 Activity 覆盖或被锁屏：系统会调用 onPause()方法，暂停当前 Activity 的执行。

（3）当前 Activity 由被覆盖状态回到前台或解锁屏：系统会调用 onResume()方法，再次进入运行状态。

（4）当前 Activity 转到新的 Activity 界面或按 Home 键回到主屏：系统会先调用 onPause()方法，然后调用 onStop()方法，进入停止状态。

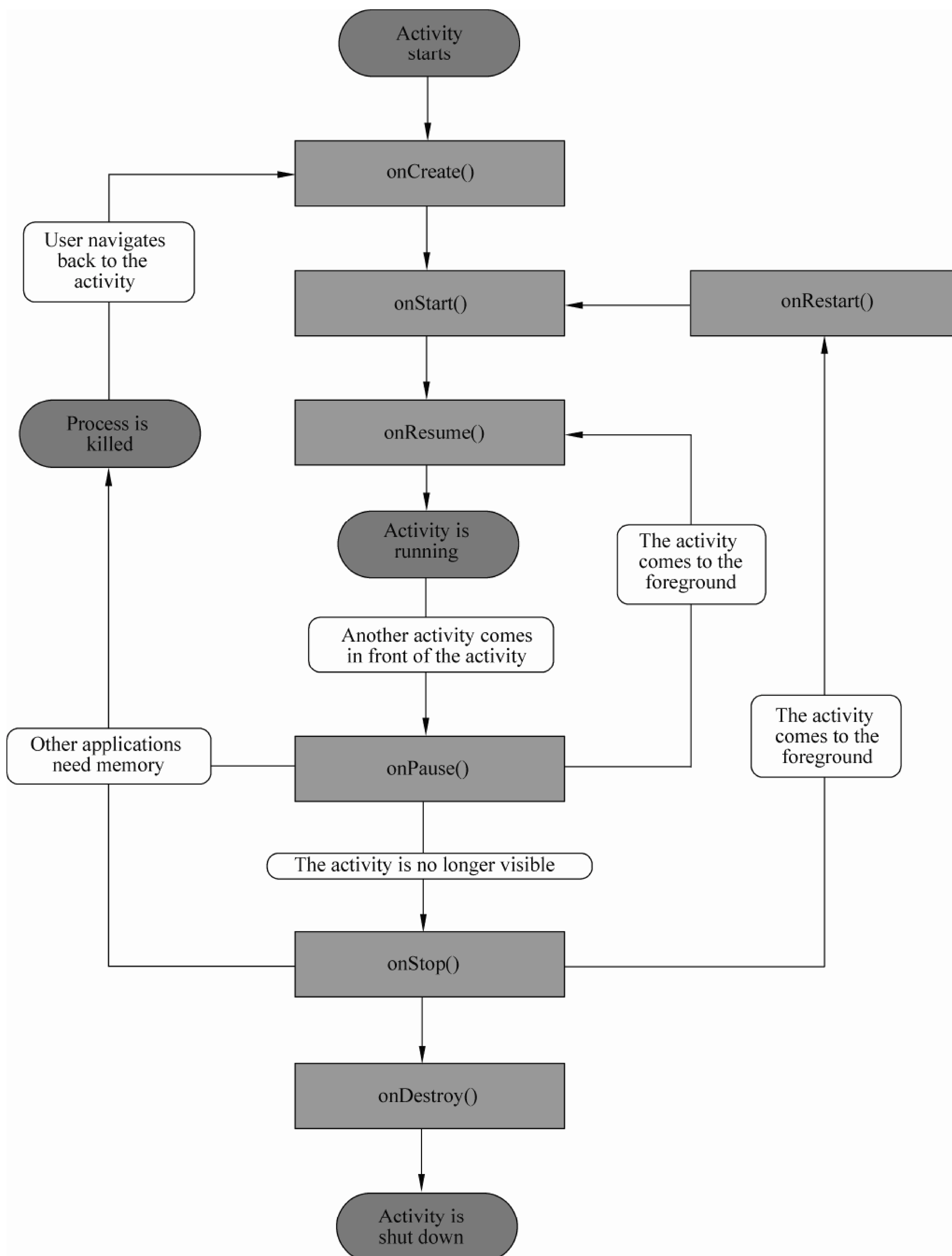


图 14-26　Activity 的生命周期

（5）用户后退到此 Activity：系统会先调用 onRestart()方法，然后调用 onStart()方法，最后调用 onResume()方法，再次进入运行状态。

（6）当前 Activity 处于被覆盖状态或者后台不可见状态，即第（2）步和第（4）步，系统内存不足，“杀死”当前 Activity。而后用户退回当前 Activity：再次调用 onCreate()方法、onStart()方法和 onResume()方法进入运行状态。

（7）用户退出当前 Activity：系统先调用 onPause()方法，然后调用 onStop()方法，最后调用 onDestory 方法，结束当前 Activity。

这个过程可以用下面的状态图来简单说明，如图 14-27 所示。

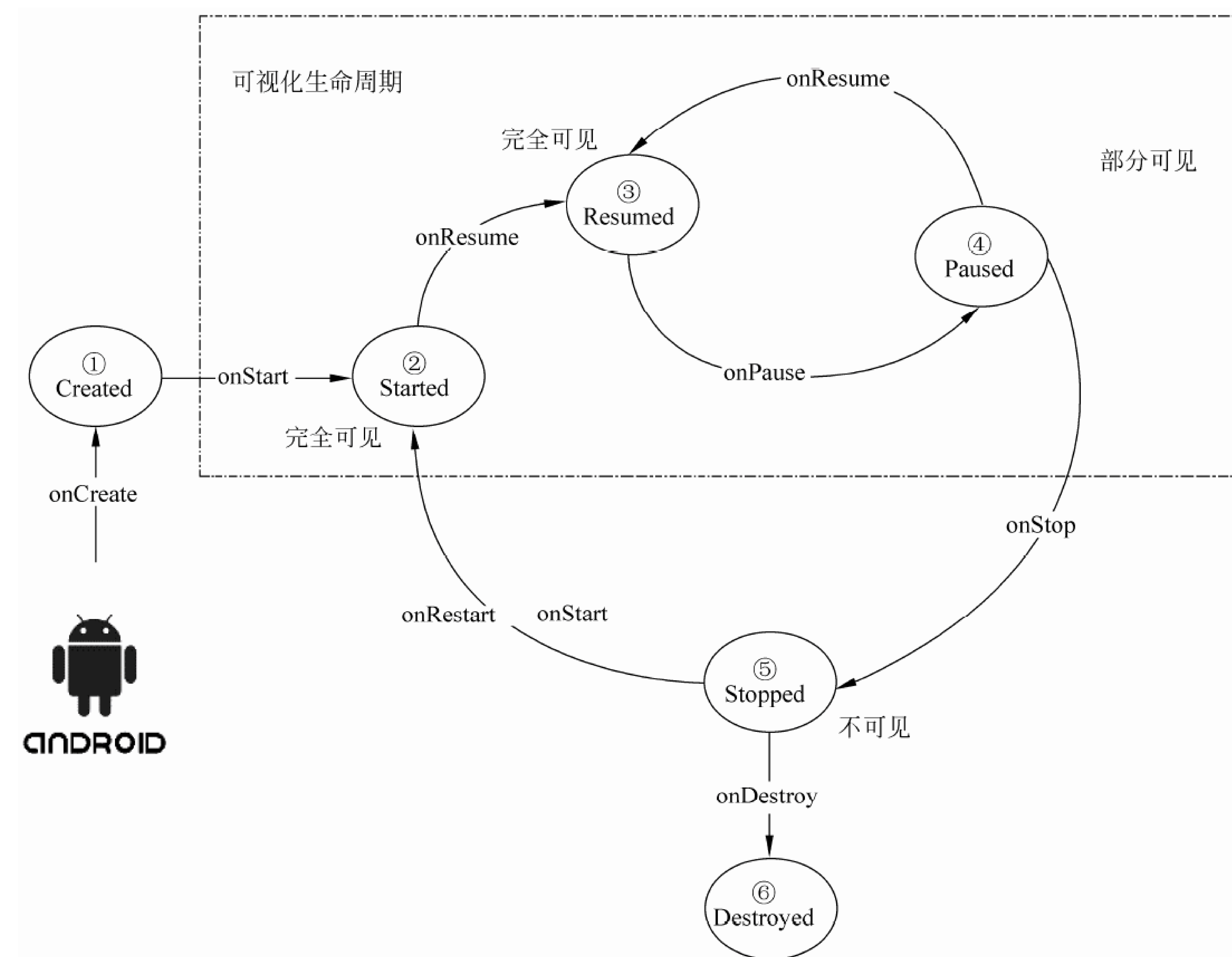


图 14-27　Activity 状态图

14.2.5　Kotlin Android Extensions 插件

在上面的 ItemListActivity.onCreate 函数中，其中的这行代码

```
setSupportActionBar(toolbar)
```

是设置支持的 ActionBar()方法。但是我们发现这里并没有使用 findViewById()方法来获取 android:id="@+id/toolbar"Toolbar 的 View 对象，之前我们可能都是这样写的：

```
Toolbar toolbar = (Toolbar)findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
```

而这里直接使用了 toolbar 这个 Toolbar 的对象变量。这是怎么做到的呢？其实是通过 Kotlin Android Extensions 插件做到的。我们在 app 目录下的 Gradle 配置文件 build.gradle 中添加了以下配置：

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
```

有了这个插件，就可以永远与 findViewById 说再见了。Kotlin Android Extensions 插件是 Kotlin 针对 Android 开发专门定制的通用插件，通过它能够以极简的无缝方式实现从 Activity、Fragment 和 View 布局组件中创建和获取视图 View。使用 Kotlin 开发 Android 大大减少了我们的样板代码。

就像上面的示例代码一样，只要在代码中直接使用这个布局组件的 ID 名称作为变量名即可，剩下的部分 Kotlin 插件会全部“搞定”。Kotlin Android Extensions 插件将会生成一些额外的代码，使我们可以在布局 XML 中直接通过 ID 获取到其 View 对象。另外，它还会生成一个本地视图缓存，当第一次使用属性时，将执行一个常规的 findViewById。但在下一次使用属性的时候，视图将从缓存中恢复，因此访问速度将更快。

只要在布局中添加一个 View，在 Activity、View、Fragment 中都可以直接用 ID 来引用这个 View。Kotlin 把 Android 编程极简的风格发挥得淋漓尽致。

我们可以通过 Kotlin 对应的字节码来更加深入地理解 Kotlin 所做的事情。Android Studio 中与 IDEA 一样提供了 Kotlin 的工具箱。在菜单栏中依次选择 Tools | Kotlin | Show Kotlin Bytecode 命令，如图 14-28 所示。

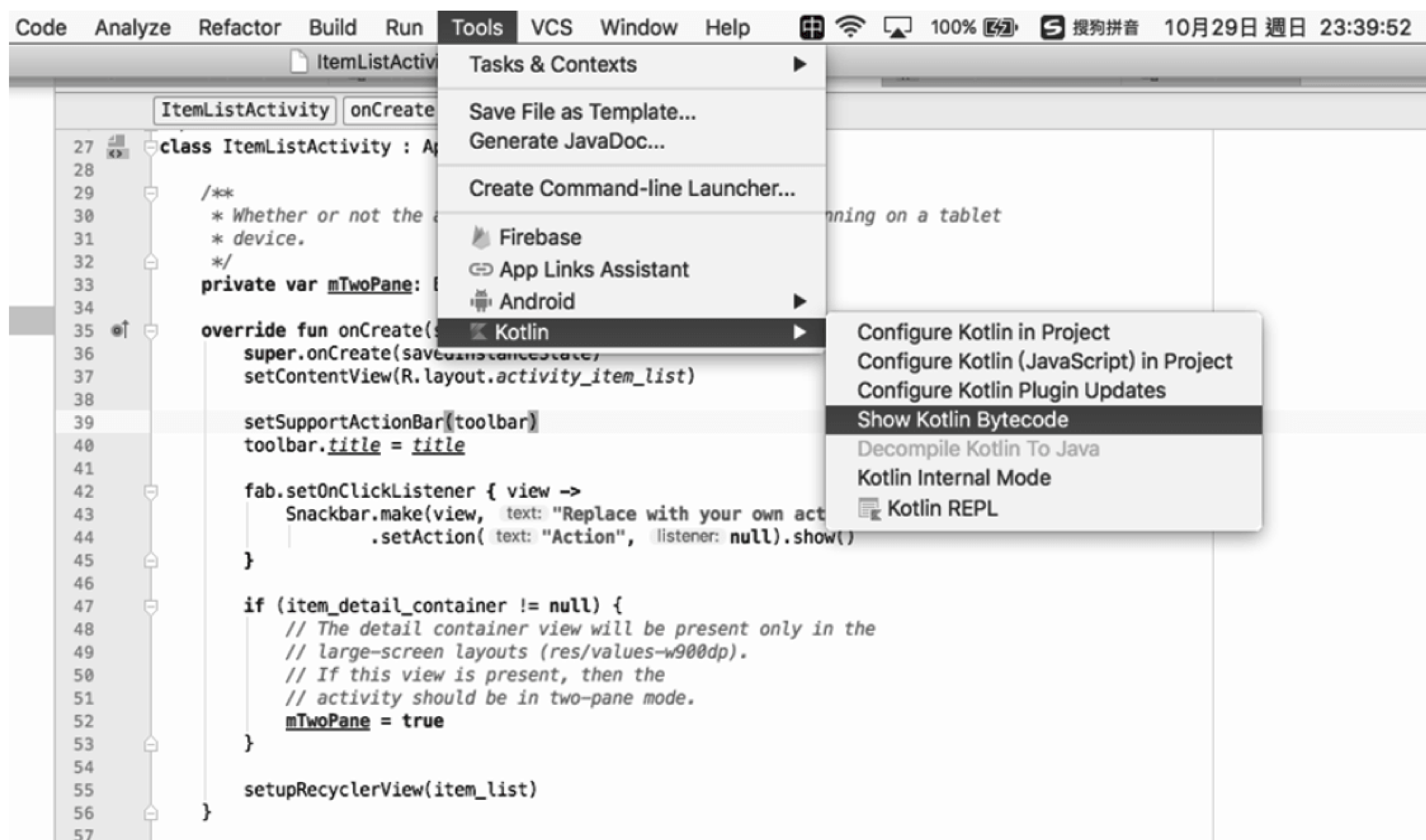


图 14-28　依次选择 Tools | Kotlin | Show Kotlin Bytecode 命令

之后将会看到如图 14-29 所示的 Kotlin Bytecode 界面。

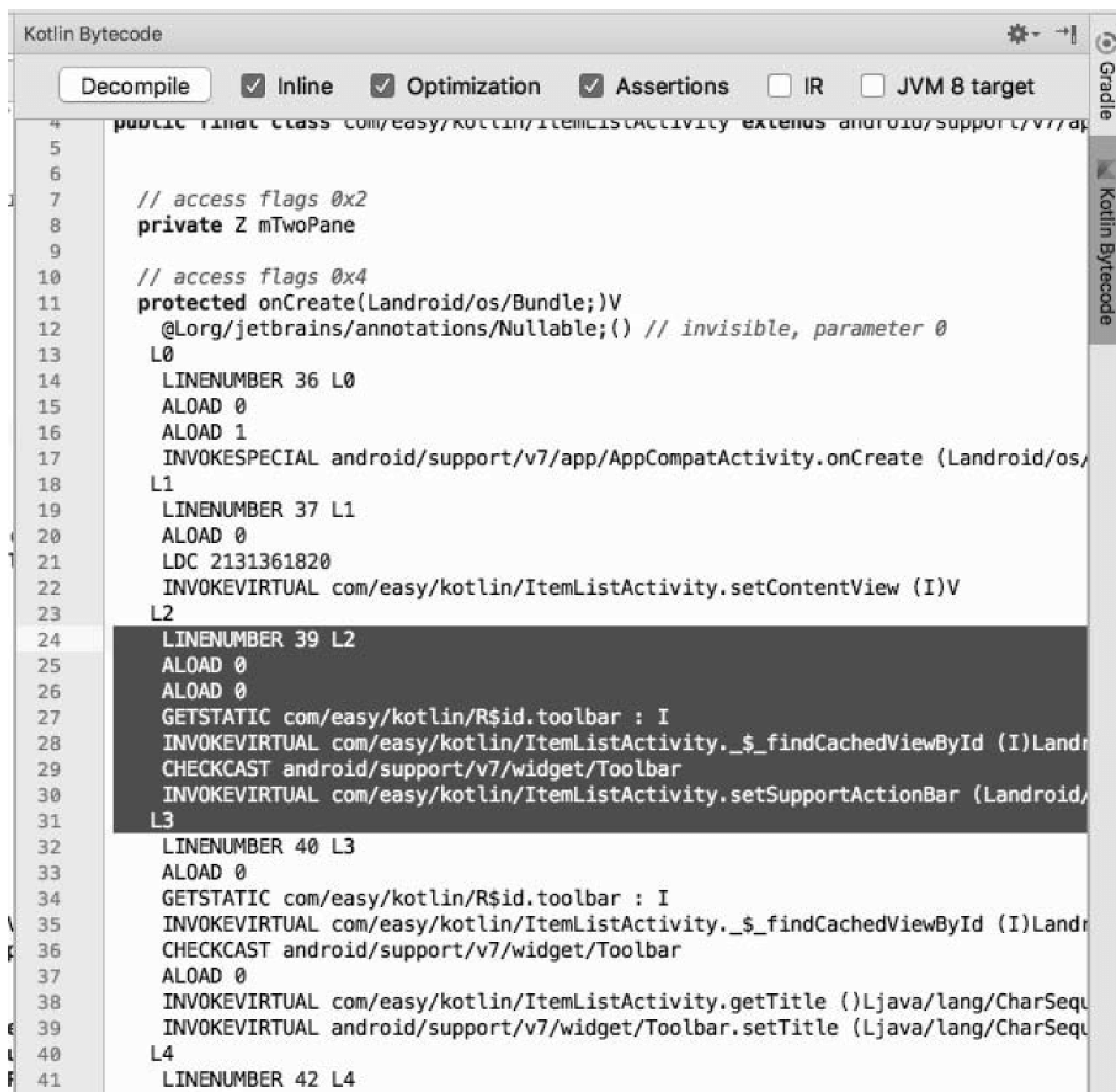


图 14-29　Kotlin Bytecode 界面

其中，下面的两行代码：

```
setSupportActionBar(toolbar)
toolbar.title = title
```

对应的字节码如下：

```
    LINENUMBER 39 L2
    ALOAD 0
    ALOAD 0
    GETSTATIC com/easy/kotlin/R$id.toolbar : I
    INVOKEVIRTUAL  com/easy/kotlin/ItemListActivity._$_findCachedViewById
(I)Landroid/view/View;
    CHECKCAST android/support/v7/widget/Toolbar
    INVOKEVIRTUAL   com/easy/kotlin/ItemListActivity.setSupportActionBar
(Landroid/support/v7/widget/Toolbar;)V
   L3
    LINENUMBER 40 L3
    ALOAD 0
    GETSTATIC com/easy/kotlin/R$id.toolbar : I
    INVOKEVIRTUAL  com/easy/kotlin/ItemListActivity._$_findCachedViewById
(I)Landroid/view/View;
    CHECKCAST android/support/v7/widget/Toolbar
    ALOAD 0
    INVOKEVIRTUAL com/easy/kotlin/ItemListActivity.getTitle ()Ljava/lang/
CharSequence;
```

```
    INVOKEVIRTUAL android/support/v7/widget/Toolbar.setTitle (Ljava/lang/
    CharSequence;)V
   L4
```

其实从字节码中

```
GETSTATIC com/easy/kotlin/R$id.toolbar : I
    INVOKEVIRTUAL com/easy/kotlin/ItemListActivity._$_findCachedViewById
```

我们已经看到了 Kotlin 所做的事情了。反编译成 Java 代码可能会看得更加清楚：

```
public final class ItemListActivity extends AppCompatActivity {
   private boolean mTwoPane;
   private HashMap _$_findViewCache;

   protected void onCreate(@Nullable Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      this.setContentView(2131361820);
      this.setSupportActionBar((Toolbar)this._$_findCachedViewById(id.toolbar));
      ((Toolbar)this._$_findCachedViewById(id.toolbar)).setTitle(this.getTitle());
      ...
   }

   public View _$_findCachedViewById(int var1) {
      if(this._$_findViewCache == null) {
         this._$_findViewCache = new HashMap();
      }

      View var2 = (View)this._$_findViewCache.get(Integer.valueOf(var1));
      if(var2 == null) {
         var2 = this.findViewById(var1);
         this._$_findViewCache.put(Integer.valueOf(var1), var2);
      }

      return var2;
   }
   ...
}
```

其中，ItemListActivity 类中 HashMap 类型的私有成员变量 $findViewCache 就是本地缓存。这里其实反映出了 Kotlin 语言设计的核心思想：通过对 Java 更高一层的封装，不仅大大简化了样板化的代码量，同时根据一些特定的可以优化的问题场景，提供了更好的性能。

同样的，上面代码中的 fab 变量：

```
fab.setOnClickListener { view ->
    Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
            .setAction("Action", null).show()
}
```

也是直接使用的布局 XML 中的 android:id="@+id/fab"：

```
<android.support.design.widget.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="@dimen/fab_margin"
    app:srcCompat="@android:drawable/ic_dialog_email" />
```

item_detail_container、setupRecyclerView(item_list)中的 item_list 都是使用了上面的方式，这样代码确实精简了许多。

上面的 activity_item_list.xml 布局中嵌套的 FrameLayout 布局配置如下：

```
<FrameLayout
    android:id="@+id/frameLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior">

    <include layout="@layout/item_list" />

</FrameLayout>
```

其中，<include layout="@layout/item_list"/>表示引用 layout 文件夹下面的 item_list.xml，item_list.xml 文件内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.RecyclerView
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/item_list"
    android:name="com.easy.kotlin.ItemListFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginLeft="16dp"
    android:layout_marginRight="16dp"
    app:layoutManager="LinearLayoutManager"
    tools:context="com.easy.kotlin.ItemListActivity"
    tools:listitem="@layout/item_list_content" />
```

而布局 item_list.xml 中的 tools:listitem="@layout/item_list_content"表示引用了 layout 文件夹下面的 item_list_content.xml 布局文件。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <TextView
        android:id="@+id/id_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/text_margin"
        android:textAppearance="?attr/textAppearanceListItem" />

    <TextView
        android:id="@+id/content"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/text_margin"
        android:textAppearance="?attr/textAppearanceListItem" />
</LinearLayout>
```

14.2.6　详情页 ItemDetailActivity

ItemDetailActivity 是 Item 详情页的 Activity，对应的 Kotlin 代码如下：

```
package com.easy.kotlin

import android.content.Intent
import android.os.Bundle
import android.support.design.widget.Snackbar
import android.support.v7.app.AppCompatActivity
import android.view.MenuItem
import kotlinx.android.synthetic.main.activity_item_detail.*

/**
 * An activity representing a single Item detail screen. This
 * activity is only used on narrow width devices. On tablet-size devices,
 * item details are presented side-by-side with a list of items * in a
   [ItemListActivity].
 */
class ItemDetailActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_item_detail)
        setSupportActionBar(detail_toolbar)

        fab.setOnClickListener { view ->
            Snackbar.make(view, "Replace with your own detail action", Snackbar.
           LENGTH_LONG)
                    .setAction("Action", null).show()
        }

        //Show the Up button in the action bar.
        supportActionBar?.setDisplayHomeAsUpEnabled(true)

        //savedInstanceState is non-null when there is fragment state
        //saved from previous configurations of this activity
        //(e.g. when rotating the screen from portrait to landscape).
        //In this case, the fragment will automatically be re-added
        //to its container so we don't need to manually add it.
        //For more information, see the Fragments API guide at:
        //
        //http://developer.android.com/guide/components/fragments.html
        //
        if (savedInstanceState == null) {
            //Create the detail fragment and add it to the activity
            //using a fragment transaction.
            val arguments = Bundle()
            arguments.putString(ItemDetailFragment.ARG_ITEM_ID,
                    intent.getStringExtra(ItemDetailFragment.ARG_ITEM_ID))
            val fragment = ItemDetailFragment()
            fragment.arguments = arguments
            //处理提交 Fragment 的事务
            supportFragmentManager.beginTransaction()
                    .add(R.id.item_detail_container, fragment)
                    .commit()
        }
```

```
    }

    override fun onOptionsItemSelected(item: MenuItem) =
            when (item.itemId) {
                android.R.id.home -> {
                    //This ID represents the Home or Up button. In the case of this
                    //activity, the Up button is shown. For
                    //more details, see the Navigation pattern on Android Design:
                    //
                    //http://developer.android.com/design/patterns/navigation.
                      html #up- vs-back

                    navigateUpTo(Intent(this, ItemListActivity::class.java))
                    true
                }
                else -> super.onOptionsItemSelected(item)
            }
}
```

UI 布局 XML 文件的 item_detail.xml 内容如下：

```
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout width="match parent"
    android:layout height="match parent"
    android:fitsSystemWindows="true"
    tools:context="com.easy.kotlin.ItemDetailActivity"
    tools:ignore="MergeRootFrame">

    <android.support.design.widget.AppBarLayout
        android:id="@+id/app_bar"
        android:layout width="match parent"
        android:layout height="@dimen/app bar height"
        android:fitsSystemWindows="true"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar">

        <android.support.design.widget.CollapsingToolbarLayout
            android:id="@+id/toolbar layout"
            android:layout_width="match_parent"
            android:layout height="match parent"
            android:fitsSystemWindows="true"
            app:contentScrim="?attr/colorPrimary"
            app:layout scrollFlags="scroll|exitUntilCollapsed"
            app:toolbarId="@+id/toolbar">

            <android.support.v7.widget.Toolbar
                android:id="@+id/detail toolbar"
                android:layout width="match parent"
                android:layout height="?attr/actionBarSize"
                app:layout_collapseMode="pin"
                app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

        </android.support.design.widget.CollapsingToolbarLayout>

    </android.support.design.widget.AppBarLayout>
```

```
    <android.support.v4.widget.NestedScrollView
        android:id="@+id/item_detail_container"
        android:layout width="match parent"
        android:layout height="match parent"
        app:layout behavior="@string/appbar scrolling view behavior" />

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout width="wrap content"
        android:layout_height="wrap_content"
        android:layout gravity="center vertical|start"
        android:layout margin="@dimen/fab margin"
        app:layout anchor="@+id/item detail container"
        app:layout_anchorGravity="top|end"
        app:srcCompat="@android:drawable/stat notify chat" />

</android.support.design.widget.CoordinatorLayout>
```

打开 item_detail.xml，可以看到设计图的 UI 效果如图 14-30 所示。

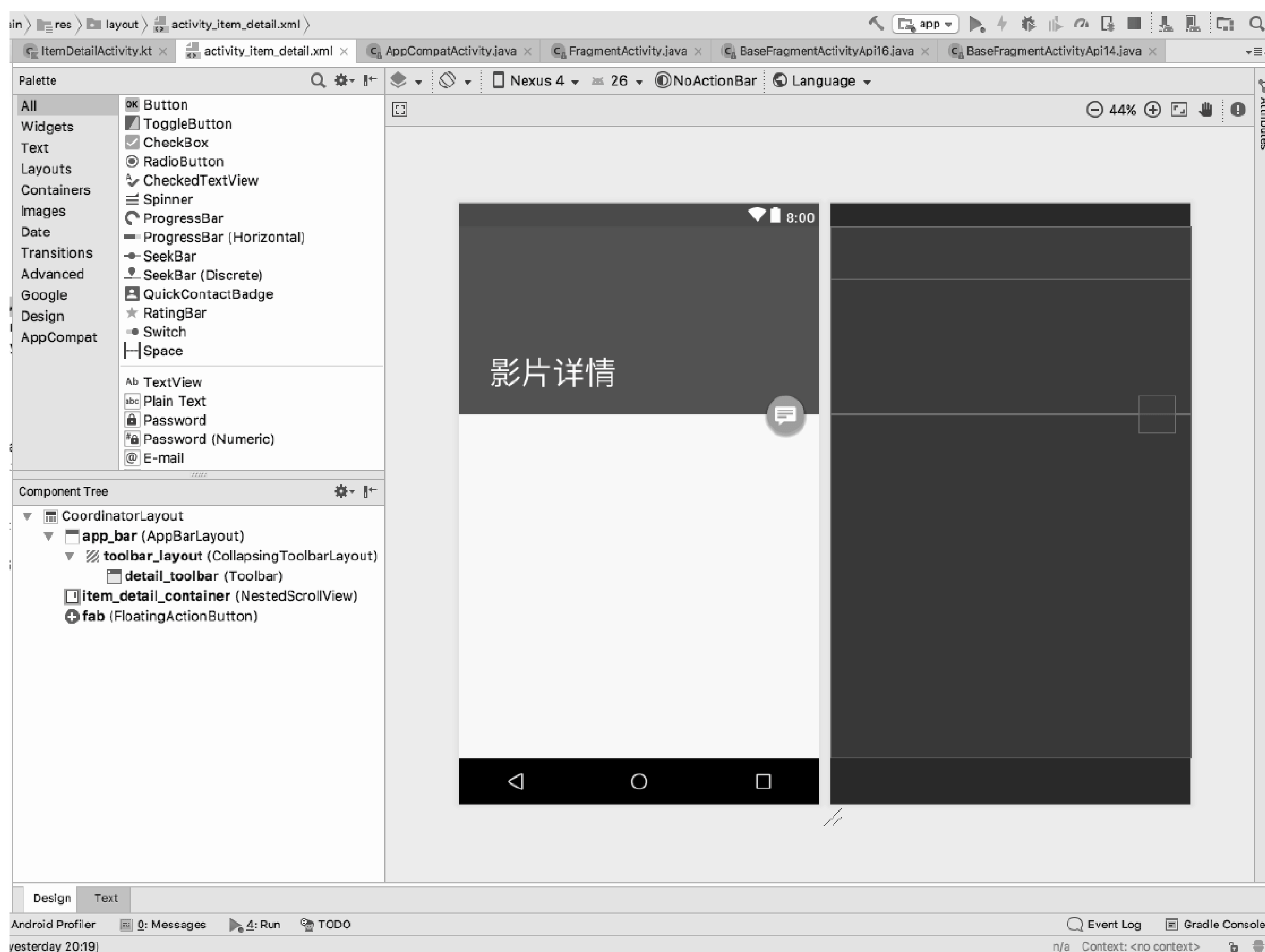


图 14-30　item_detail.xml 设计图的 UI 效果

可以看到，详情页的布局主要有 3 大块，分别是 AppBarLayout、NestedScrollView 和 FloatingActionButton。

在 ItemDetailActivity 的 onCreate()函数里的

```
setContentView(R.layout.activity_item_detail)
```

设置详情页 ItemDetailActivity 的显示界面中使用 activity_item_detail.xml 布局文件进行布局：

```
setSupportActionBar(detail_toolbar)
```

设置详情页的 android.support.v7.widget.Toolbar 控件布局。

下面来看在 ItemDetailActivity 中创建 ItemDetailFragment 的过程。代码如下：

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    if (savedInstanceState == null) {
        //Create the detail fragment and add it to the activity
        //using a fragment transaction.
        val arguments = Bundle()
        arguments.putString(ItemDetailFragment.ARG ITEM ID,
                intent.getStringExtra(ItemDetailFragment.ARG_ITEM_ID))
        val fragment = ItemDetailFragment()
        fragment.arguments = arguments
        supportFragmentManager.beginTransaction()
                .add(R.id.item_detail_container, fragment)
                .commit()
    }
}
```

（1）首先判断当前 savedInstanceState 是否为空。如果为空，执行步骤（2）。

（2）创建 ItemDetailFragment()对象，并设置其 Bundle 信息（Fragment 中的成员变量 mArguments）如下：

```
val arguments = Bundle()
arguments.putString(ItemDetailFragment.ARG ITEM ID,
        intent.getStringExtra(ItemDetailFragment.ARG ITEM ID))
val fragment = ItemDetailFragment()
fragment.arguments = arguments
```

（3）通过 supportFragmentManager 添加 Fragment 与布局空间的映射关系。

```
supportFragmentManager.beginTransaction()
        .add(R.id.item detail container, fragment)
        .commit()
```

其中，supportFragmentManager 用来获取能管理和当前 Activity 有关联的 Fragment 的 FragmentManager，使用 supportFragmentManager 可以向 Activity 状态中添加一个 Fragment。

上面代码中的 R.id.item_detail_container 对应的布局是一个 NestedScrollView，代码如下：

```
<android.support.v4.widget.NestedScrollView
    android:id="@+id/item detail container"
    android:layout width="match parent"
    android:layout height="match parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" />
```

UI 界面的设计效果图如图 14-31 所示。

最后需要注意的是，如果当前 Activity 在前面已经保存了 Fragment 状态的数据，那么

savedInstanceState 的值就是非空的，这个时候我们就不需要再去手工创建 Fragment 对象保存到当前的 Activity 中了。因为当我们的 Activty 被异常销毁时，Activity 会对自身状态进行保存（这里包含了我们添加的 Fragment）。而在 Activity 被重新创建时，又会对我们之前保存的 Fragment 进行恢复。

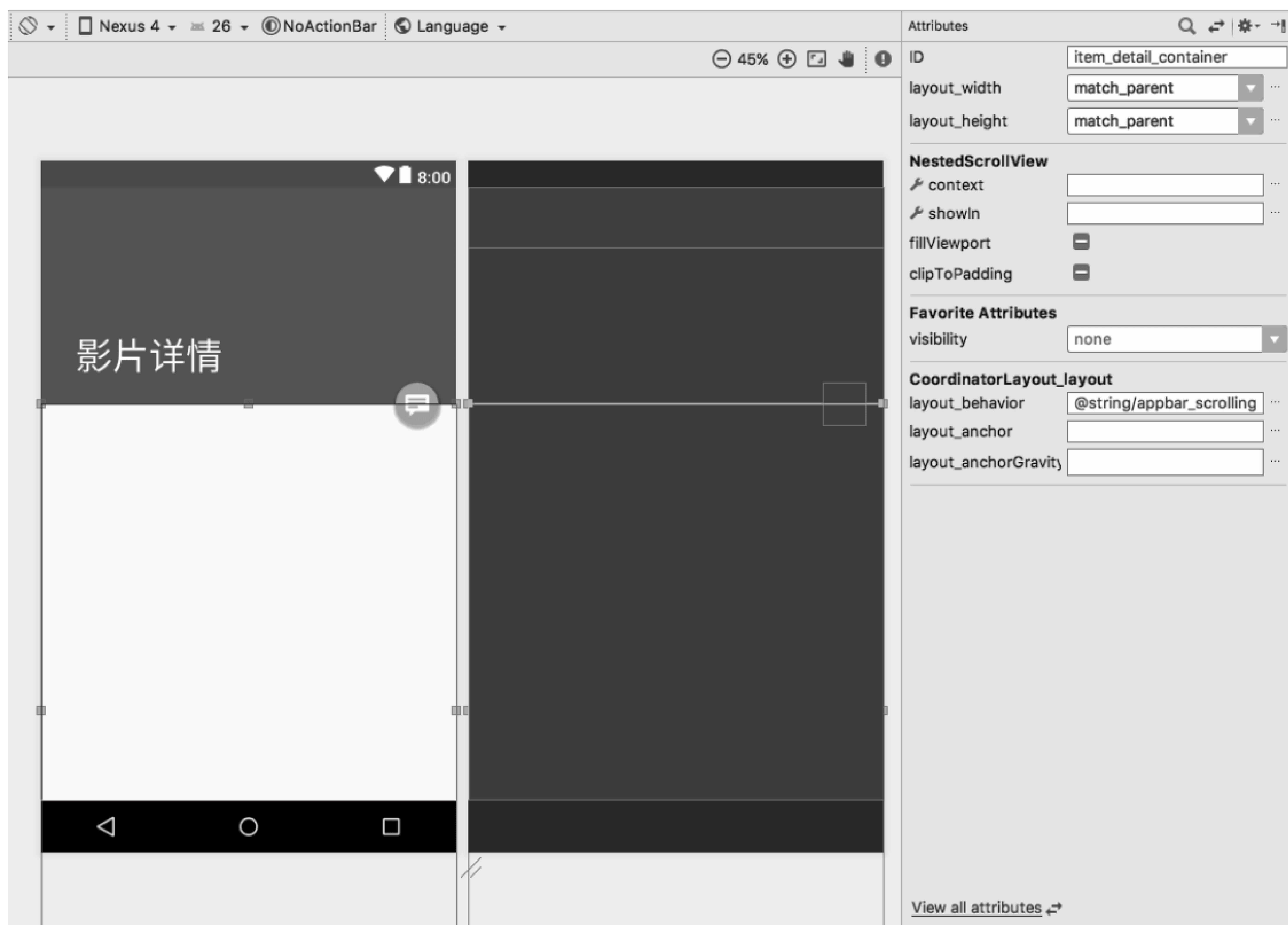


图 14-31　NestedScrollView 代码布局效果图

所以，添加 Fragment 前千万要记得检查是否有保存的 Activity 状态。如果没有状态保存，说明 Activity 是第 1 次被创建，我们需要添加 Fragment；如果有状态保存，说明 Activity 刚刚出现过异常被销毁过，之前的 Fragment 会被恢复，我们不用再添加 Fragment。

14.2.7　碎片事务类 FragmentTransaction

前面的代码中使用了 FragmentTransaction 的 add()方法，该方法签名如下：

```
public abstract FragmentTransaction add(@IdRes int containerViewId,
Fragment fragment);
```

其中，参数 containerViewId 为传入 Activity 中某个视图容器的 ID。如果 containerViewId 传入 0，则这个 Fragment 不会被放置在一个容器中。请注意，不要认为 Fragment 没添加进来，其实我们只是添加了一个没有视图的 Fragment 而已，这个 Fragment 可以用来做一些类似于 Service 的后台工作。

FragmentTransaction 常用的 API 如表 14-1 所示。

表 14-1　FragmentTransaction常用的API方法

API 方法	说　　明
add(int containerViewId, Fragment fragment, String tag)	向 Activity state 中添加一个 Fragment。参数 containerViewId 一般为传入 Activity 中某个视图容器的 ID。如果 containerViewId 传入 0，则这个 Fragment 不会被放置在一个容器中。添加 Fragment 前应检查是否有保存的 Activity 状态
remove(Fragment fragment)	移除一个已经存在的 Fragment。Fragment 被 remove 后，Fragment 的生命周期会一直执行完 onDetach，之后 Fragment 的实例也会从 Fragment Manager 中被移除
replace(int containerViewId, Fragment fragment)	替换一个已被添加进视图容器的 Fragment。之前添加的 Fragment 会在 replace 时被视图容器移除
addToBackStack(String name)	记录已提交的事务（transation），可用于回退操作。参数 name 是这次回退操作的一个名称（或标识），不需要时可以传入 null
show(Fragment fragment)	隐藏一个存在的 Fragment
hide(Fragment fragment)	显示一个以前被隐藏过的 Fragment。Fragment 被 hide/show，仅仅是隐藏/显示 Fragment 的视图，不会有任何生命周期方法的调用。在 Fragment 中重写 onHiddenChanged()方法可以对 Fragment 的 hide 和 show 状态进行监听
attach(Fragment fragment)	重新关联一个 Fragment（当这个 Fragment 的 detach 执行之后）。当 Fragment 被 detach 后，执行 attach 操作，会让 Fragment 从 onCreateView 开始执行，一直执行到 onResume。attach 无法像 add 一样单独使用，因为如果单独使用会抛异常。该方法存在的意义是对 detach 后的 Fragment 进行界面恢复
detach(Fragment fragment)	分离指定 Fragment 的 UI 视图。当 Fragment 被 detach 后，Fragment 的生命周期执行完 onDestroyView()方法就终止了，这意味着 Fragment 的实例并没有被销毁，只是 UI 界面被移除了（注意和 remove 是有区别的）
setCustomAnimations(int enter, int exit)	为 Fragment 的进入/退出设置指定的动画资源
commit()	提交事务。安排一个针对该事务的提交。提交并没有立刻发生，会安排到在主线程下次准备好的时候来执行
commitNow()	同步提交这个事务。任何被添加的 Fragment 都将会被初始化，并将它们完全带入它们的生命周期状态。 使用 commitNow()时不能进行添加回退栈的操作，如果使用 addToBackStack(String)将会抛出一个 IllegalStateException 的异常

下面介绍 ItemDetailFragment。

ItemDetailFragment 表示单个 Item 的详细信息。此片段在双窗格模式（在平板电脑上）包含在 ItemListActivity 中，在手机上则是包含在 ItemDetailActivity 中。其 Kotlin 代码如下：

```
package com.easy.kotlin

import android.os.Bundle
import android.support.v4.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
```

```
import com.easy.kotlin.dummy.DummyContent
import kotlinx.android.synthetic.main.activity_item_detail.*
import kotlinx.android.synthetic.main.item_detail.view.*

class ItemDetailFragment : Fragment() {
    /**
     * 测试数据 dummy content this fragment is presenting.
     */
    private var mItem: DummyContent.DummyItem? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        if (arguments.containsKey(ARG_ITEM_ID)) {
            //加载数据
            mItem = DummyContent.ITEM_MAP[arguments.getString(ARG_ITEM_ID)]
            mItem?.let {
                //给 toolbar_layout 布局设置标题
                activity.toolbar_layout?.title = it.content
            }
        }
    }

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val rootView = inflater.inflate(R.layout.item_detail, container, false)

        //在 TextView 中显示测试数据文本
        mItem?.let {
            rootView.item_detail.text = it.details
        }

        return rootView
    }

    companion object
{       /**
         * The fragment argument representing the item ID that this fragment
         * represents.
         */
        const val ARG_ITEM_ID = "item_id"
    }
}
```

在 onCreate()中，activity.toolbar_layout?.title=it.content 这行代码是给详情页 ToolBar 的大标题赋值。

```
<android.support.design.widget.AppBarLayout
    android:id="@+id/app_bar"
    android:layout_width="match_parent"
    android:layout_height="@dimen/app_bar_height"
    android:fitsSystemWindows="true"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar">

    <android.support.design.widget.CollapsingToolbarLayout
        android:id="@+id/toolbar_layout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
```

```
        android:fitsSystemWindows="true"
        app:contentScrim="?attr/colorPrimary"
        app:layout_scrollFlags="scroll|exitUntilCollapsed"
        app:toolbarId="@+id/toolbar">

        <android.support.v7.widget.Toolbar
            android:id="@+id/detail_toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:layout_collapseMode="pin"
            app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

    </android.support.design.widget.CollapsingToolbarLayout>

</android.support.design.widget.AppBarLayout>
```

对应的 UI 效果图如图 14-32 所示。



图 14-32　AppBarLayout 的 UI 界面效果图

在 onCreateView()中，rootView.item_detail.text = it.details 这行代码对应的布局是单个 Item 的详情展示 TextView 视图，其布局 XML 代码中的 item_detail.xml 代码如下：

```
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/item_detail"
    style="?android:attr/textAppearanceLarge"
    android:layout_width="match_parent"
```

```
    android:layout height="match parent"
    android:padding="16dp"
    android:textIsSelectable="true"
    tools:context="com.easy.kotlin.ItemDetailFragment" />
```

UI 效果图如图 14-33 所示。

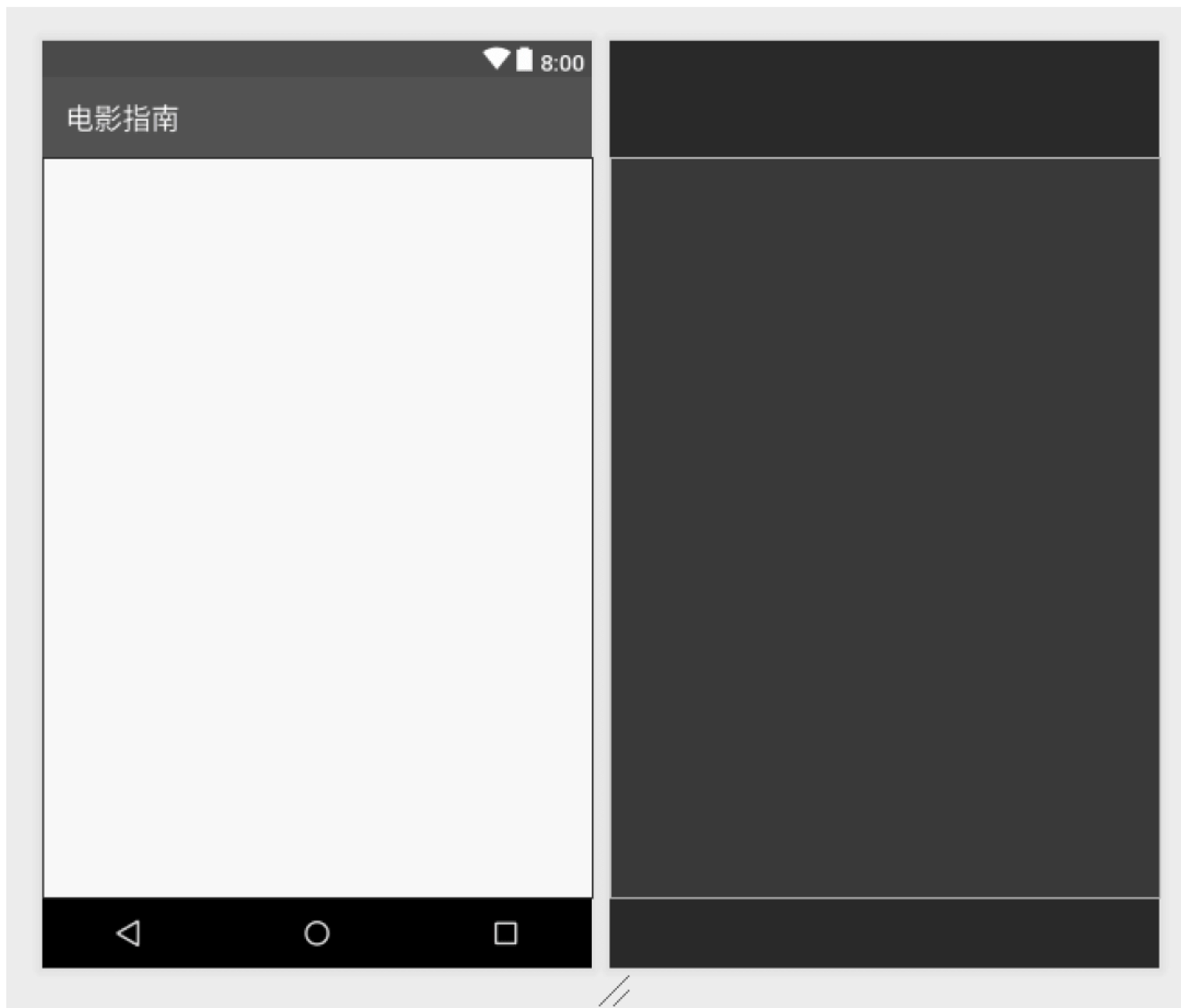


图 14-33　item_detail.xml 的布局 UI 效果图

14.2.8　Fragment 生命周期

Fragment 必须嵌入在 Activity 中才能生存，其生命周期也直接受宿主 Activity 生命周期的影响。例如，若宿主 Activity 处于 pause 状态，则它所管辖的 Fragment 也将进入 pause 状态。而当 Activity 处于 resume 状态时，则可以独立地控制每一个 Fragment，如添加或删除等。为了创建 Fragment，需要继承一个 Fragment 类，并实现 Fragment 的生命周期回调方法，如 onCreate()、onStart()、onPause()和 onStop()等。事实上，若需要在一个应用中加入 Fragment，只需要将原来的 Activity 替换为 Fragment，并将 Activity 的生命周期回调方法简单地改为 Fragment 的生命周期回调方法即可。Fragment 的生命周期示意图如图 14-34 所示。

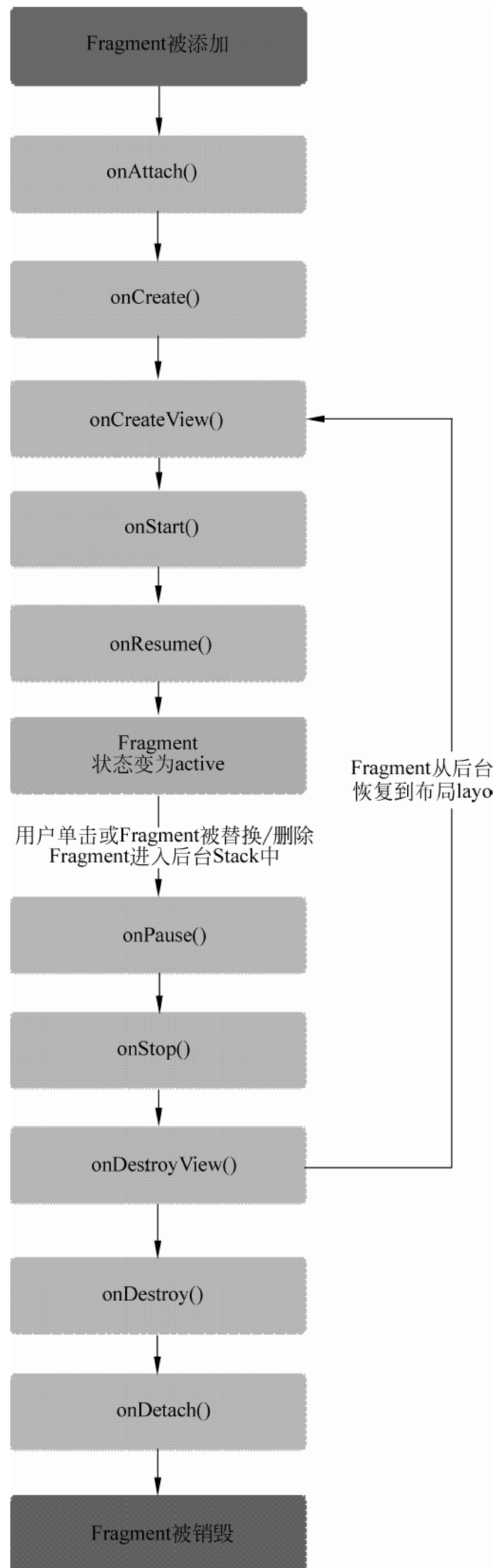


图 14-34　Fragment 的生命周期示意图

另外，Fragment 与 Activity 的生命周期对比如图 14-35 所示。

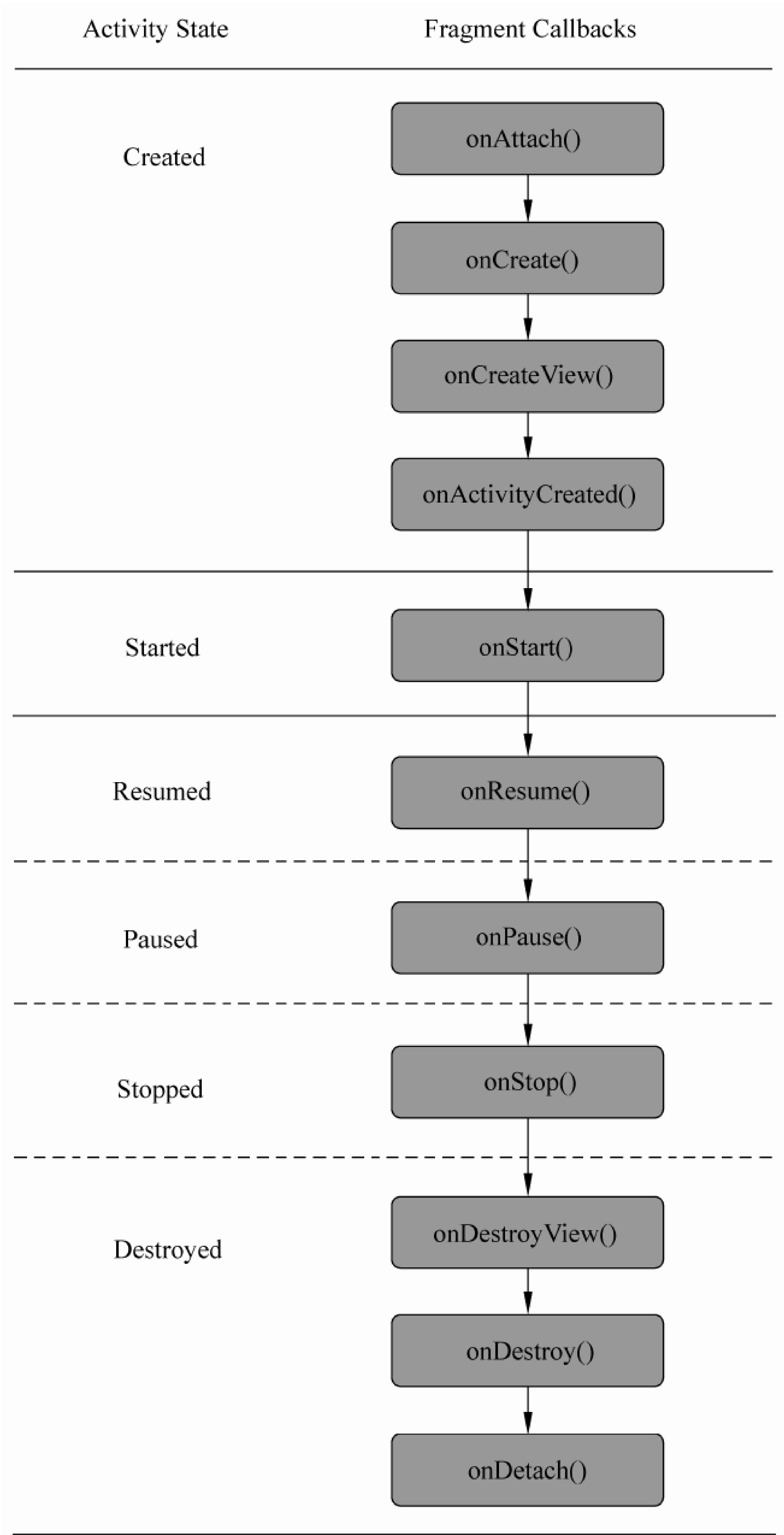


图 14-35　Fragment 与 Activity 的生命周期对比

（1）当一个 Fragment 被创建的时候，会依次经历以下状态：OnAttach()、onCreate()、onCreateView()和 onActivityCreated()。

（2）当这个 Fragment 对用户可见的时候，会经历 onStart()和 onResume()两个状态。

（3）当这个 fragment 进入“后台模式”的时候，会经历 onPause()和 onStop()两个状态。

（4）当这个 Fragment 被销毁了（或者持有它的 Activity 被销毁了），会经历以下状态：

onPause()、onStop()、onDestroyView()和 onDetach()。

（5）就像 Activity 一样，在以下状态中，可以使用 Bundle 对象保存一个 Fragment 对象：onCreate()、onCreateView()和 onActivityCreated()。

（6）Fragments 的大部分状态都和 Activity 很相似，但 Fragment 有一些新的状态，这些新状态如下。

- onAttached()：当 Fragment 和 Activity 关联之后，调用这个方法；
- onCreateView()：创建 Fragment 中的视图时，调用这个方法；
- onActivityCreated()：当 Activity 的 onCreate()方法被返回之后，调用这个方法；
- onDestroyView()：当 Fragment 中的视图被移除的时候，调用这个方法；
- onDetach()：当 Fragment 和 Activity 分离的时候，调用这个方法。

一般来说，在 Fragment 中应至少重写下面 3 个生命周期方法：

- onCreate()：当创建 Fragment 实例时，系统回调的方法。在该方法中，需要对一些必要的组件进行初始化，以保证这个组件的实例在 Fragment 处于 pause 或 stop 状态时仍然存在。
- onCreateView()：当第一次在 Fragment 上绘制 UI 时，系统回调的方法。该方法返回一个 View 对象，该对象表示 Fragment 的根视图；若 Fragment 不需要展示视图，则该方法可以返回 null。
- onPause()：当用户离开 Fragment 时回调的方法（并不意味着该 Fragment 被销毁）。在该方法中，可以对 Fragment 的数据信息做一些持久化的保存工作，因为用户可能不再返回这个 Fragment。

大多数情况下，需要重写上述 3 个方法，有时还需要重写其他生命周期方法。

当执行一个 Fragment 事务时，也可以将该 Fragment 加入到一个由宿主 Activity 管辖的后退栈中，并由 Activity 记录加入到后退栈的 Fragment 信息，按下后退键可以将 Fragment 从后退栈中一次弹出。

将 Fragment 添加至 Activity 的视图布局中有两种方式：一种是使用 Fragment 标签加入。Fragment 的父视图应是一个 ViewGroup；另一种使用代码动态加入，并将一个 ViewGroup 作为 Fragment 的容器。

为了方便，继承下面这些特殊的 Fragment 可以简化其初始化过程。

- DialogFragment：可展示一个悬浮对话框。使用该类创建的对话框可以很好地替换由 Activity 类中的方法创建的对话框，因为可以像管理其他 Fragment 一样管理 DialogFragment——它们都被压入由宿主 Activity 管理的 Fragment 栈中，这可以很方便地找回已被压入栈中的 Fragment。
- ListFragment：可以展示一个内置的 AdapterView，该 AdapterView 由一个 Adapter 管理，如 SimpleCursorAdapter。ListFragment 类似于 ListActivity，它提供了大量的用于管理 ListView 的方法，如回调方法 onListItemClick()，其用于处理单击项事件。
- PreferenceFragment：可以展示层级嵌套的 Preference 对象列表。PreferenceFragment 类似于 PreferenceActivity，该类一般用于为应用程序编写设置页面。

Fragment绑定UI布局需要重写onCreateView()方法，该方法返回一个 View 视图对象，代码如下：

```
class ItemDetailFragment : Fragment() {

    private var mItem: DummyContent.DummyItem? = null

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val rootView = inflater.inflate(R.layout.item_detail, container, false)

        //Show the dummy content as text in a TextView.
        mItem?.let {
            rootView.item_detail.text = it.details
        }

        return rootView
    }
}
```

其中，val rootView=inflater.inflate(R.layout.item_detail,container, false)这一行代码中的 inflater.inflate 是用于填充布局的,这是布局填充器 LayoutInflater 类的方法。通常我们加载布局的任务都是在 Activity 中调用 setContentView()方法来完成的。其实 setContentView()方法的内部也是使用 LayoutInflater 类来加载布局的，相关的代码在 android.support.v7.app.AppCompatDelegateImplV9 中。

```
@Override
public void setContentView(int resId) {
    ensureSubDecor();
    ViewGroup contentParent = (ViewGroup) mSubDecor.findViewById(android.R.id.
    content);
    contentParent.removeAllViews();
    LayoutInflater.from(mContext).inflate(resId, contentParent);
    mOriginalWindowCallback.onContentChanged();
}
```

在实际开发中，LayoutInflater 类还是非常有用的，它的作用类似于 findViewById()。不同点是 LayoutInflater 是用来找 res\layout\下的 XML 布局文件并实例化（填充布局）；而 findViewById()是找 XML 布局文件下的具体 widget 控件（如 Button、TextView 等）并实例化。

LayoutInflater 具体作用说明如下：

- 对于一个没有被载入或者想要动态载入的界面，都需要使用 LayoutInflater.inflate()来载入；
- 对于一个已经载入的界面，可以使用 Activiyt.findViewById()方法来获得其中的界面元素。

注意： 若继承的 Fragment 是 ListFragment，onCreateView()方法已默认返回了 ListView 对象，因此无须再重写该方法。

有关 Fragment 的更多信息，请参见“Fragment API 指南”：http://developer.android.com/guide/components/fragments.html。

14.2.9 测试数据类 DummyContent

在 DummyContent 类中构造了我们在 ListActivity 中展示的测试数据。代码如下：

```
package com.easy.kotlin.dummy

import java.util.ArrayList
import java.util.HashMap

object DummyContent {

    val ITEMS: MutableList<DummyItem> = ArrayList()
    val ITEM_MAP: MutableMap<String, DummyItem> = HashMap()

    private val COUNT = 25

    init {
        //Add some sample items.
        for (i in 1..COUNT) {
            addItem(createDummyItem(i))
        }
    }

    private fun addItem(item: DummyItem) {
        ITEMS.add(item)
        ITEM_MAP.put(item.id, item)
    }

    private fun createDummyItem(position: Int): DummyItem {
        return DummyItem(position.toString(), "Item " + position, makeDetails
        (position))
    }

    private fun makeDetails(position: Int): String {
        val builder = StringBuilder()
        builder.append("Details about Item: ").append(position)
        for (i in 0..position - 1) {
            builder.append("\nMore details information here.")
        }
        return builder.toString()
    }

    data class DummyItem(val id: String, val content: String, val details: String){
        override fun toString(): String = content
    }
}
```

至此，我们已经了解了怎样使用 Android Studio 3.0 创建一个带 ListActivity 和 Fragment 列表及其详情页的方法，同时学习了 Activity 和 Fragment 的基本用法。

下面我们来实现后端 API 的接入与数据的展现。

14.2.10 创建领域对象类 Movie

创建领域对象类 Movie。我们使用 Kotlin 中的数据类来实现，代码如下：

```
data class Movie(val id: String, val title: String, val overview: String,
val posterPath: String) {
    override fun toString(): String {
        return "Movie(id='$id', title='$title', overview='$overview', posterPath
        ='$posterPath')"
    }
}
```

其中的 id、title、overview、posterPath 分别与 JSON 中的 key 对应。接下来是 JSON 数据的解析。

14.2.11　JSON 数据解析

我们调用的 API 是：

```
val VOTE_AVERAGE_API =
"http://api.themoviedb.org//3/discover/movie? certification country=US&cer-
tification=R&sort by=vote average.desc&api key=7e55a88ece9f03408b895a96
c1487979"
```

其数据返回如下：

```
{
  "page": 1,
  "total results": 10350,
  "total pages": 518,
  "results": [
    {
      "vote count": 28,
      "id": 138878,
      "video": false,
      "vote average": 10,
      "title": "Fatal Mission",
      "popularity": 3.721883,
      "poster path": "/u351Rsqu5nd36ZpbWxIpd3CpbJW.jpg",
      "original language": "en",
      "original title": "Fatal Mission",
      "genre_ids": [
        10752,
        28,
        12
      ],
      "backdrop_path": "/wNq5uqVDT7a5G1b97ffYf4hxzYz.jpg",
      "adult": false,
      "overview": "A CIA Agent must rely on reluctant help from a female spy
       in the North Vietnam jungle in order to pass through enemy lines.",
      "release date": "1990-07-25"
    },
    ...
    ]
}
```

我们使用 fastjson 来解析这个 JSON 数据。在 app 文件夹下的 build.gradle 中添加以下依赖：

```
dependencies {
    ...

    //https://mvnrepository.com/artifact/com.alibaba/fastjson
    compile group: 'com.alibaba', name: 'fastjson', version: '1.2.39'
}
```

解析代码如下：

```
val jsonstr = URL(VOTE_AVERAGE_API).readText(Charset.defaultCharset())
try {
    val obj = JSON.parse(jsonstr) as Map<*, *>
     val dataArray = obj.get("results") as JSONArray
    //TODO

    } catch (ex: Exception) {
}
```

然后把这个 dataArray 放到我们的 MovieContent 对象中。

```
dataArray.forEachIndexed { index, it ->
        val title = (it as Map<*, *>).get("title") as String
        val overview = it.get("overview") as String
        val poster_path = it.get("poster_path") as String
        addMovie(Movie(index.toString(), title, overview, getPosterUrl(poster_path)))
}
```

其中，addMovie 代码如下：

```
object MovieContent {
val MOVIES: MutableList<Movie> = ArrayList()
val MOVIE_MAP: MutableMap<String, Movie> = HashMap()
...
private fun addMovie(movie: Movie) {
    MOVIES.add(movie)
    MOVIE_MAP.put(movie.id, movie)
}
}
```

然后再分别新建 MovieDetailActivity、MovieDetailFragment、MovieListActivity 及 activity_movie_list.xml、activity_movie_detail.xml、movie_detail.xml、movie_list.xml、movie_list_content.xml 文件，下面分别对它们进行介绍。

14.2.12 电影列表页面

MovieListActivity 是电影列表页面的 Activity，代码如下：

```
package com.easy.kotlin

import android.content.Intent
import android.os.Bundle
import android.support.v7.app.AppCompatActivity
import android.support.v7.widget.RecyclerView
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.ImageView
```

```
import android.widget.TextView
import com.easy.kotlin.bean.MovieContent
import com.easy.kotlin.util.HttpUtil
import kotlinx.android.synthetic.main.activity_movie_detail.*
import kotlinx.android.synthetic.main.activity_movie_list.*
import kotlinx.android.synthetic.main.movie_list.*
import kotlinx.android.synthetic.main.movie_list_content.view.*

class MovieListActivity : AppCompatActivity() {

    private var mTwoPane: Boolean = false

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_movie_list)

        setSupportActionBar(toolbar)
        toolbar.title = title

        if (movie_detail_container != null) {
            mTwoPane = true
        }
        setupRecyclerView(movie_list)
    }

    private fun setupRecyclerView(recyclerView: RecyclerView) {
        recyclerView.adapter = SimpleItemRecyclerViewAdapter(this, MovieContent.
        MOVIES, mTwoPane)
    }

    class SimpleItemRecyclerViewAdapter(private val mParentActivity: MovieList-
    Activity,
                                        private val mValues: List<MovieContent.Movie>,
                                        private val mTwoPane: Boolean) :
RecyclerView.Adapter<SimpleItemRecyclerViewAdapter.ViewHolder>() {

        private val mOnClickListener: View.OnClickListener

        init {
            mOnClickListener = View.OnClickListener { v ->
                val item = v.tag as MovieContent.Movie
                if (mTwoPane) {
                    val fragment = MovieDetailFragment().apply {
                        arguments = Bundle()
                        arguments.putString(MovieDetailFragment.ARG_MOVIE_ID,
                        item.id)
                    }
                    mParentActivity.supportFragmentManager
                            .beginTransaction()
                            .replace(R.id.movie_detail_container, fragment)
                            .commit()
                } else {
                    val intent = Intent(v.context, MovieDetailActivity::class.
                    java).apply {
                        putExtra(MovieDetailFragment.ARG_MOVIE_ID, item.id)
                    }
                    v.context.startActivity(intent)
                }
            }
        }
```

```
        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
        ViewHolder {
            val view =
                    LayoutInflater
                            .from(parent.context)
                            .inflate(R.layout.movie_list_content, parent, false)
            return ViewHolder(view)
        }

        override fun onBindViewHolder(holder: ViewHolder, position: Int) {
            val item = mValues[position]
            holder.mIdView.text = item.id
            holder.mTitle.text = item.title
            holder.mMoviePosterImageView.setImageBitmap(HttpUtil.getBitmap
            FromURL(item.posterPath))

            with(holder.itemView) {
                tag = item
                setOnClickListener(mOnClickListener)
            }
        }

        override fun getItemCount(): Int {
            return mValues.size
        }

        inner class ViewHolder(mView: View) : RecyclerView.ViewHolder(mView) {
            val mIdView: TextView = mView.id_text
            val mTitle: TextView = mView.title
            val mMoviePosterImageView: ImageView = mView.movie_poster_image

        }
    }
}
```

对应的布局文件分别如下。

activity_movie_list.xml 文件如下：

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context="com.easy.kotlin.MovieListActivity">

    <android.support.design.widget.AppBarLayout
        android:id="@+id/app_bar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

    </android.support.design.widget.AppBarLayout>
```

```
    <FrameLayout
        android:id="@+id/frameLayout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior">

        <include layout="@layout/movie_list" />
    </FrameLayout>

</android.support.design.widget.CoordinatorLayout>
```

movie_list.xml 文件如下：

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.RecyclerView
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/movie_list"
    android:name="com.easy.kotlin.MovieListFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginLeft="16dp"
    android:layout_marginRight="16dp"
    app:layoutManager="LinearLayoutManager"
    tools:context="com.easy.kotlin.MovieListActivity"
    tools:listitem="@layout/movie_list_content" />
```

movie_list_content.xml 文件如下：

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="320dp"
    android:layout_gravity="center"
    android:layout_margin="0dp"
    android:clickable="true"
    android:foreground="?attr/selectableItemBackground"
    android:orientation="horizontal">

    <TextView
        android:id="@+id/id_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/text_margin"
        android:textAppearance="?attr/textAppearanceListItem" />

    <ImageView
        android:id="@+id/movie_poster_image"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scaleType="centerCrop" />

    <View
        android:id="@+id/title_background"
        android:layout_width="match_parent"
        android:layout_height="48dp"
        android:layout_gravity="bottom"
        android:alpha="0.8"
        android:background="@color/colorPrimaryDark"
        android:gravity="center" />
```

```
    <TextView
        android:id="@+id/title"
        android:layout width="match parent"
        android:layout height="48dp"
        android:layout gravity="bottom"
        android:gravity="center"
        android:paddingLeft="@dimen/activity horizontal margin"
        android:paddingRight="@dimen/activity horizontal margin"
        android:textColor="@android:color/white"
        android:textSize="12sp" />

</FrameLayout>
```

电影列表整体布局的 UI 效果图如图 14-36 所示。

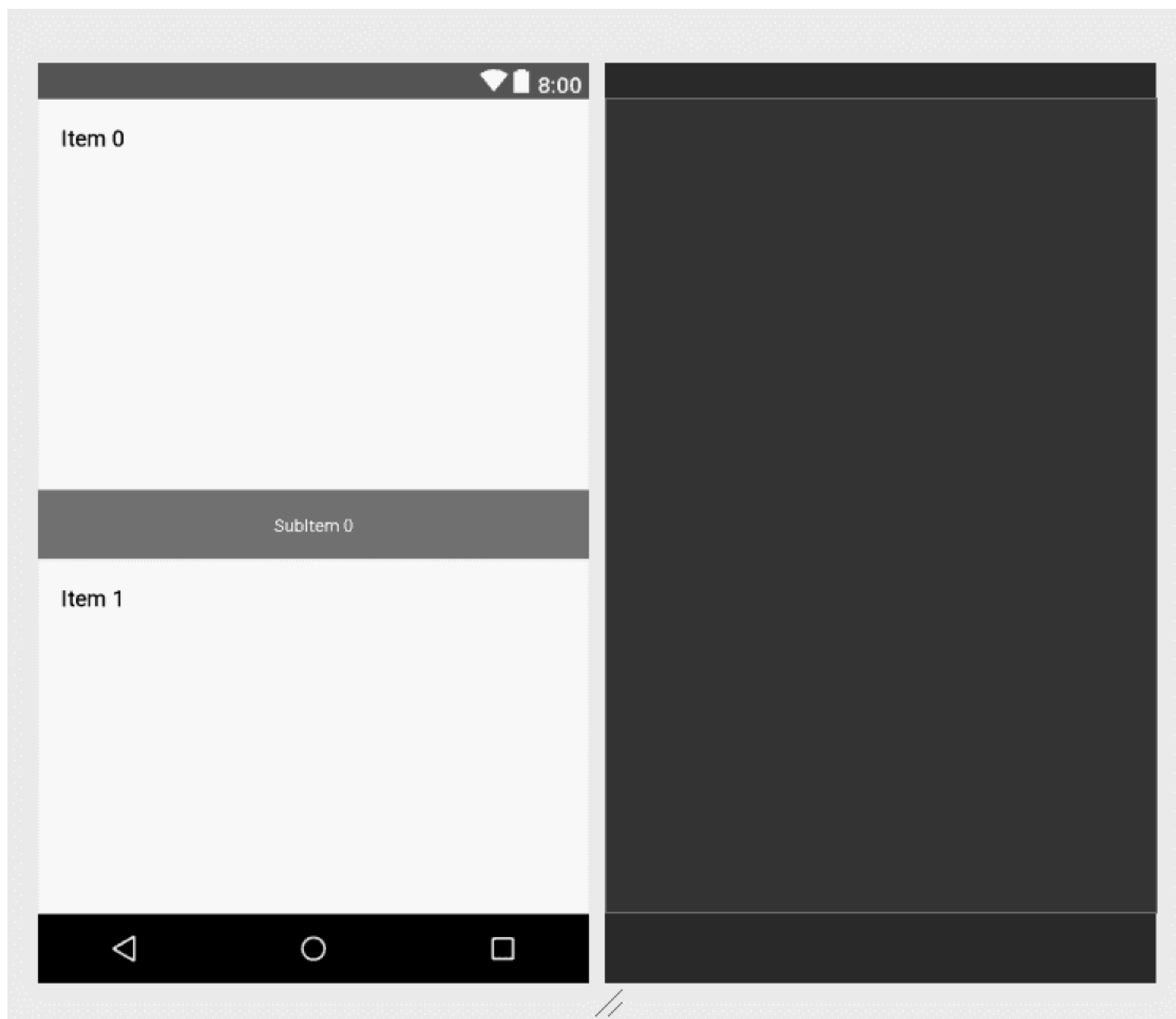


图 14-36　电影列表的整体布局

14.2.13　视图数据适配器 ViewAdapter

在创建 MovieListActivity 过程中需要展示响应的数据，这些数据由 ViewAdapter 来承载，对应的代码如下：

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_movie_list)

    setSupportActionBar(toolbar)
    toolbar.title = title

    if (movie_detail_container != null) {
        mTwoPane = true
    }

    setupRecyclerView(movie_list)
}

private fun setupRecyclerView(recyclerView: RecyclerView) {
    recyclerView.adapter = SimpleItemRecyclerViewAdapter(this, MovieContent.
    MOVIES, mTwoPane)
}
```

在上面的代码中定义了一个继承 RecyclerView.Adapter 的 SimpleItemRecyclerViewAdapter 类，来装载 View 中要显示的数据，实现数据与视图的解耦。View 要显示的数据从 Adapter 里获取并展现出来。Adapter 负责把真实的数据适配成一个个 View，也就是说 View 要显示什么数据，取决于 Adapter 里的数据。

14.2.14　视图中图像的展示

在函数 SimpleItemRecyclerViewAdapter.onBindViewHolder()中，设置 View 组件与 Model 数据的绑定。其中的电影海报是图片，所以我们的布局文件中使用了 ImageView，对应的布局文件是 movie_list_content.xml，代码如下:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="320dp"
    android:layout_gravity="center"
    android:layout_margin="0dp"
    android:clickable="true"
    android:foreground="?attr/selectableItemBackground"
    android:orientation="horizontal">

    <TextView
        android:id="@+id/id_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/text_margin"
        android:textAppearance="?attr/textAppearanceListItem" />

    <ImageView
        android:id="@+id/movie_poster_image"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scaleType="centerCrop" />

    <View
        android:id="@+id/title_background"
        android:layout_width="match_parent"
```

```
        android:layout_height="48dp"
        android:layout_gravity="bottom"
        android:alpha="0.8"
        android:background="@color/colorPrimaryDark"
        android:gravity="center" />

    <TextView
        android:id="@+id/title"
        android:layout_width="match_parent"
        android:layout_height="48dp"
        android:layout_gravity="bottom"
        android:gravity="center"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        android:textColor="@android:color/white"
        android:textSize="12sp" />

</FrameLayout>
```

UI 设计效果图如图 14-37 所示。

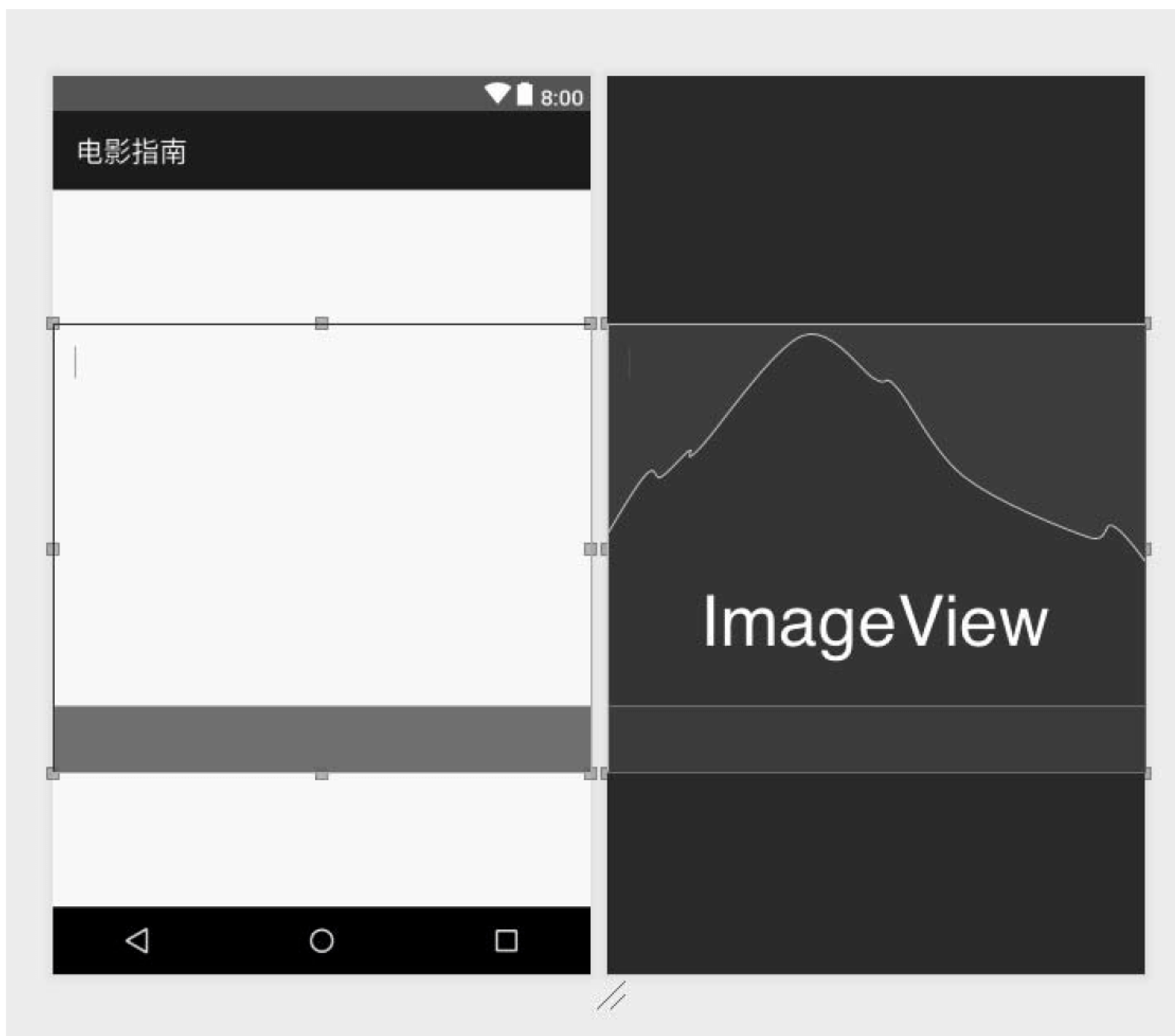


图 14-37　movie_list_content.xml 布局 UI 效果图

列表中图片的视图组件标签是 ImageView，布局的 XML 代码如下：

```
<ImageView
    android:id="@+id/movie_poster_image"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:scaleType="centerCrop" />
```

这里是根据图片的 URL 来展示图片，ImageView 类有个 setImageBitmap 方法，可以直接设置 Bitmap 图片数据。见下面的代码：

```
holder.mMoviePosterImageView.setImageBitmap(HttpUtil.getBitmapFromURL
(item.posterPath))
```

而通过 URL 获取 Bitmap 图片数据的代码是：

```
object HttpUtil {
    fun getBitmapFromURL(src: String): Bitmap? {
        try {
            val url = URL(src)
            val input = url.openStream()
            val myBitmap = BitmapFactory.decodeStream(input)
            return myBitmap
        } catch (e: Exception) {
            e.printStackTrace()
            return null
        }
    }
}
```

14.2.15　电影详情页面

MovieDetailActivity 文件（电影详情页面）代码如下：

```
package com.easy.kotlin

import android.content.Intent
import android.os.Bundle
import android.support.design.widget.Snackbar
import android.support.v7.app.AppCompatActivity
import android.view.MenuItem
import kotlinx.android.synthetic.main.activity_movie_detail.*

class MovieDetailActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_movie_detail)
        setSupportActionBar(detail_toolbar)

        fab.setOnClickListener { view ->
            Snackbar.make(view, "Replace with your own detail action", Snackbar.
           LENGTH_LONG)
                    .setAction("Action", null).show()
        }
```

```
        supportActionBar?.setDisplayHomeAsUpEnabled(true)
        if (savedInstanceState == null) {
            val arguments = Bundle()
            arguments.putString(MovieDetailFragment.ARG MOVIE ID,
                    intent.getStringExtra(MovieDetailFragment.ARG MOVIE ID))
            val fragment = MovieDetailFragment()
            fragment.arguments = arguments
            supportFragmentManager.beginTransaction()
                    .add(R.id.movie detail container, fragment)
                    .commit()
        }
    }

    override fun onOptionsItemSelected(item: MenuItem) =
            when (item.itemId) {
                android.R.id.home -> {
                    navigateUpTo(Intent(this, MovieListActivity::class.java))
                    true
                }
                else -> super.onOptionsItemSelected(item)
            }
}
```

其中，详情页的布局 XML 文件是 activity_item_detail.xml，其代码如下：

```
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout height="match parent"
    android:fitsSystemWindows="true"
    tools:context="com.easy.kotlin.ItemDetailActivity"
    tools:ignore="MergeRootFrame">

    <android.support.design.widget.AppBarLayout
        android:id="@+id/app_bar"
        android:layout width="match parent"
        android:layout height="@dimen/app bar height"
        android:fitsSystemWindows="true"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar">

        <android.support.design.widget.CollapsingToolbarLayout
            android:id="@+id/toolbar layout"
            android:layout_width="match_parent"
            android:layout height="match parent"
            android:fitsSystemWindows="true"
            app:contentScrim="?attr/colorPrimary"
            app:layout_scrollFlags="scroll|exitUntilCollapsed"
            app:toolbarId="@+id/toolbar">

            <android.support.v7.widget.Toolbar
                android:id="@+id/detail_toolbar"
                android:layout width="match parent"
                android:layout height="?attr/actionBarSize"
                app:layout collapseMode="pin"
                app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />
```

```
        </android.support.design.widget.CollapsingToolbarLayout>

    </android.support.design.widget.AppBarLayout>

    <android.support.v4.widget.NestedScrollView
        android:id="@+id/item_detail_container"
        android:layout width="match parent"
        android:layout height="match parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior" />

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout height="wrap content"
        android:layout gravity="center vertical|start"
        android:layout margin="@dimen/fab margin"
        app:layout_anchor="@+id/item_detail_container"
        app:layout anchorGravity="top|end"
        app:srcCompat="@android:drawable/stat notify chat" />

</android.support.design.widget.CoordinatorLayout>
```

我们把电影详情的 Fragment 的展示放到 NestedScrollView 中：

```
<android.support.v4.widget.NestedScrollView
    android:id="@+id/movie detail container"
    android:layout width="match parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" />
```

电影详情的 Fragment 代码是 MovieDetailFragment：

```
package com.easy.kotlin

import android.os.Bundle
import android.support.v4.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.easy.kotlin.bean.MovieContent
import com.easy.kotlin.util.HttpUtil
import kotlinx.android.synthetic.main.activity_movie_detail.*
import kotlinx.android.synthetic.main.movie_detail.view.*

class MovieDetailFragment : Fragment() {
    private var mItem: MovieContent.Movie? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        if (arguments.containsKey(ARG_MOVIE_ID)) {

            mItem = MovieContent.MOVIE MAP[arguments.getString(ARG MOVIE ID)]
            mItem?.let {
                activity.toolbar_layout?.title = it.title
            }
```

```
        }
    }

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        //绑定 movieDetailView
        val movieDetailView = inflater.inflate(R.layout.movie_detail, container,
        false)
        mItem?.let {
            movieDetailView.movie_poster_image.setImageBitmap(HttpUtil.
            getBitmapFromURL(it.posterPath))
            movieDetailView.movie_overview.text = "影片简介:  ${it.overview}"
            movieDetailView.movie_vote_count.text = "打分次数: ${it.vote_count}"
            movieDetailView.movie_vote_average.text = "评分: ${it.vote_average}"
            movieDetailView.movie_release_date.text = "发行日期: ${it.release_date}"
        }

        return movieDetailView
    }

    companion object {
        const val ARG_MOVIE_ID = "movie_id"
    }
}
```

其中的 R.layout.movie_detail 布局文件 movie_detail.xml 如下：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools=http://schemas.android.com/tools
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_gravity="center"
    android:layout_margin="0dp"
    android:clickable="true"
    android:foreground="?attr/selectableItemBackground"
    android:orientation="vertical">

    <TextView
        android:id="@+id/movie_release_date"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="16dp"
        android:textIsSelectable="true"
        android:textSize="18sp"
        tools:context="com.easy.kotlin.MovieDetailFragment" />

    <ImageView
        android:id="@+id/movie_poster_image"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_centerVertical="true"
        android:fitsSystemWindows="true"
        android:scaleType="fitCenter" />

    <TextView
        android:id="@+id/movie_overview"
```

```
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="16dp"
        android:textIsSelectable="true"
        android:textSize="18sp"
        tools:context="com.easy.kotlin.MovieDetailFragment" />

    <TextView
        android:id="@+id/movie_vote_average"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="16dp"
        android:textIsSelectable="true"
        android:textSize="18sp"
        tools:context="com.easy.kotlin.MovieDetailFragment" />

    <TextView
        android:id="@+id/movie_vote_count"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="16dp"
        android:textIsSelectable="true"
        android:textSize="18sp"
        tools:context="com.easy.kotlin.MovieDetailFragment" />

</LinearLayout>
```

14.2.16　电影源数据的获取

首先定义一个 MovieContent 对象类来存储从 API 获取的数据，代码如下：

```
package com.easy.kotlin.bean

import android.os.StrictMode
import com.alibaba.fastjson.JSON
import com.alibaba.fastjson.JSONArray
import java.net.URL
import java.nio.charset.Charset
import java.util.*


object MovieContent {

    val MOVIES: MutableList<Movie> = ArrayList()
    val MOVIE_MAP: MutableMap<String, Movie> = HashMap()

    val VOTE_AVERAGE_API = "http://api.themoviedb.org//3/discover/movie?
    sort_by=popularity.desc&api_key=7e55a88ece9f03408b895a96c1487979&page=1"

    init {
        val policy = StrictMode.ThreadPolicy.Builder().permitAll().build()
        StrictMode.setThreadPolicy(policy)
        initMovieListData()
    }
```

```
    private fun initMovieListData() {

        val jsonstr = URL(VOTE_AVERAGE_API).readText(Charset.defaultCharset())
        try {
            val obj = JSON.parse(jsonstr) as Map<*, *>
            val dataArray = obj.get("results") as JSONArray

            dataArray.forEachIndexed { index, it ->
                val title = (it as Map<*, *>).get("title") as String
                val overview = it.get("overview") as String
                val poster_path = it.get("poster_path") as String
                val vote_count = it.get("vote_count").toString()
                val vote_average = it.get("vote_average").toString()
                val release_date = it.get("release_date").toString()
                addMovie(Movie(id = index.toString(),
                        title = title,
                        overview = overview,
                        vote_count = vote_count,
                        vote_average = vote_average,
                        release_date = release_date,
                        posterPath = getPosterUrl(poster_path)))
            }

        } catch (ex: Exception) {
            ex.printStackTrace()
        }
    }

    private fun addMovie(movie: Movie) {
        MOVIES.add(movie)
        MOVIE_MAP.put(movie.id, movie)
    }

    fun getPosterUrl(posterPath: String): String {
        return "https://image.tmdb.org/t/p/w185_and_h278_bestv2$posterPath"
    }

    data class Movie(val id: String,
                     val title: String,
                     val overview: String,
                     val vote_count: String,
                     val vote_average: String,
                     val release_date: String,
                     val posterPath: String)
}
```

在 Android 4.0 之后默认的线程模式是不允许在主线程中访问网络的。为了演示效果，我们在访问网络代码前，把 ThreadPolicy 设置为允许运行访问网络。

```
val policy = StrictMode.ThreadPolicy.Builder().permitAll().build()
StrictMode.setThreadPolicy(policy)
```

然后使用一个 data class Movie 来存储电影对象数据。

```
data class Movie(val id: String,
                 val title: String,
                 val overview: String,
```

```
                val vote_count: String,
                val vote_average: String,
                val release_date: String,
                val posterPath: String)
```

14.2.17　配置 AndroidManifest.xml

最后配置 AndroidManifest.xml 文件内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.easy.kotlin">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
        <activity
            android:name=".MovieListActivity"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".MovieDetailActivity"
            android:label="@string/title_movie_detail"
            android:parentActivityName=".MovieListActivity"
            android:theme="@style/AppTheme.NoActionBar">
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value="com.easy.kotlin.MovieListActivity" />
        </activity>
    </application>

    <uses-permission android:name="android.permission.INTERNET" />

</manifest>
```

因为我们要访问网络，所以需要添加以下配置：

```
<uses-permission android:name="android.permission.INTERNET" />
```

14.2.18　打包安装测试

再次打包安装并运行程序，电影列表页面如图 14-38 所示。

点击进入电影详情页，如图 14-39 所示。



图 14-38　电影列表页面



图 14-39　电影详情页

14.3　本章小结

Android 中经常出现的空引用、API 的冗余样板式代码等都是驱动我们转向 Kotlin 语言的动力。另外，Kotlin 的 Android 视图 DSL Anko 可以让我们从繁杂的 XML 视图配置文件中解放出来。

我们可以像在 Java 中一样方便地使用 Android 开发流行的库，如 Butter Knife、Realm、RecyclerView 等。当然，使用 Kotlin 集成这些库进行 Android 开发，既能够直接使用我们之前的开发库，又能够从 Java 语言、Android API 的限制中解脱出来，这不得不说是一件好事。未来，Android 领域将是 Kotlin 的天下。

本章工程源码地址是 https://github.com/Android-Kotlin/MovieGuideDB。